

Fig. 1a

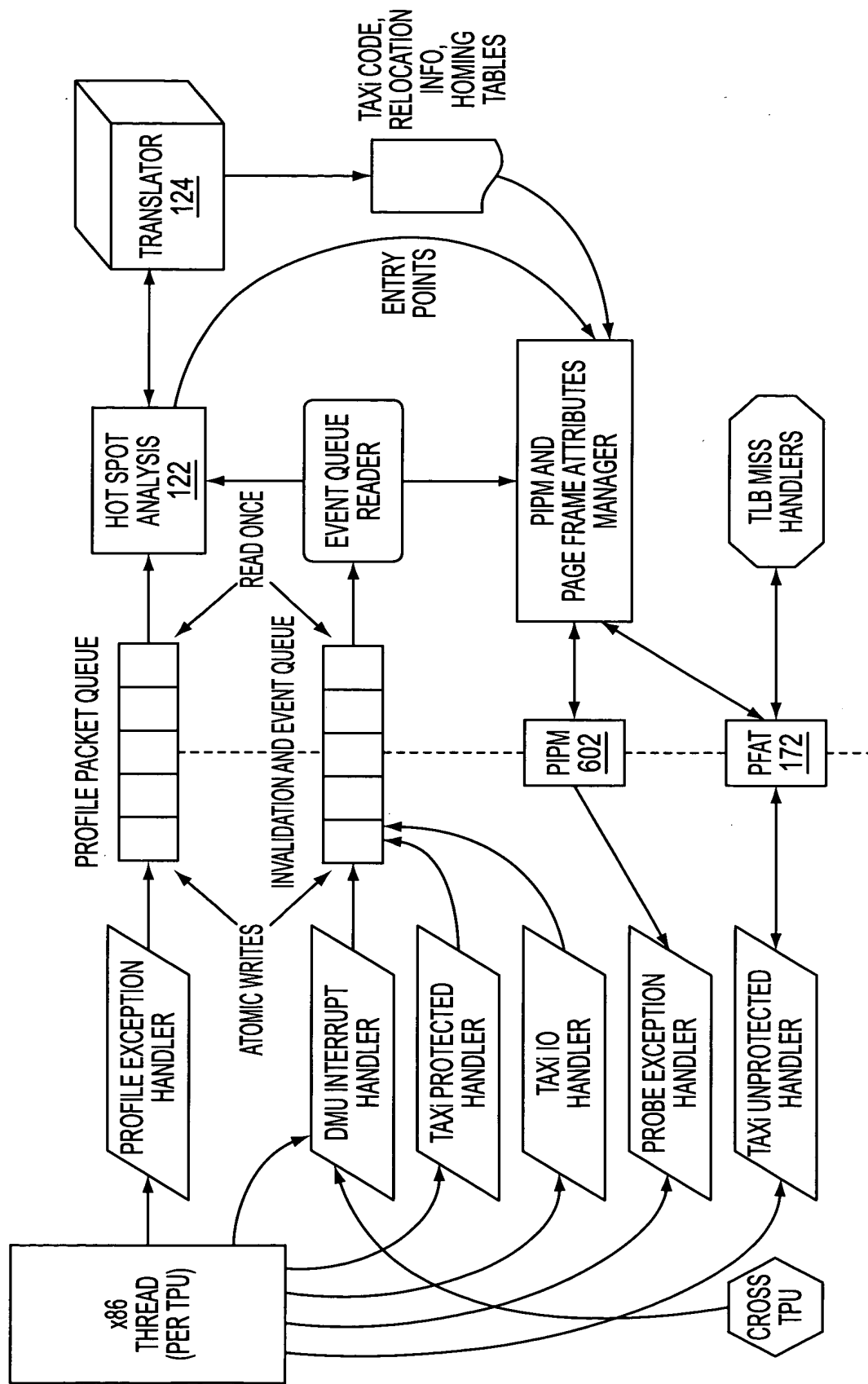
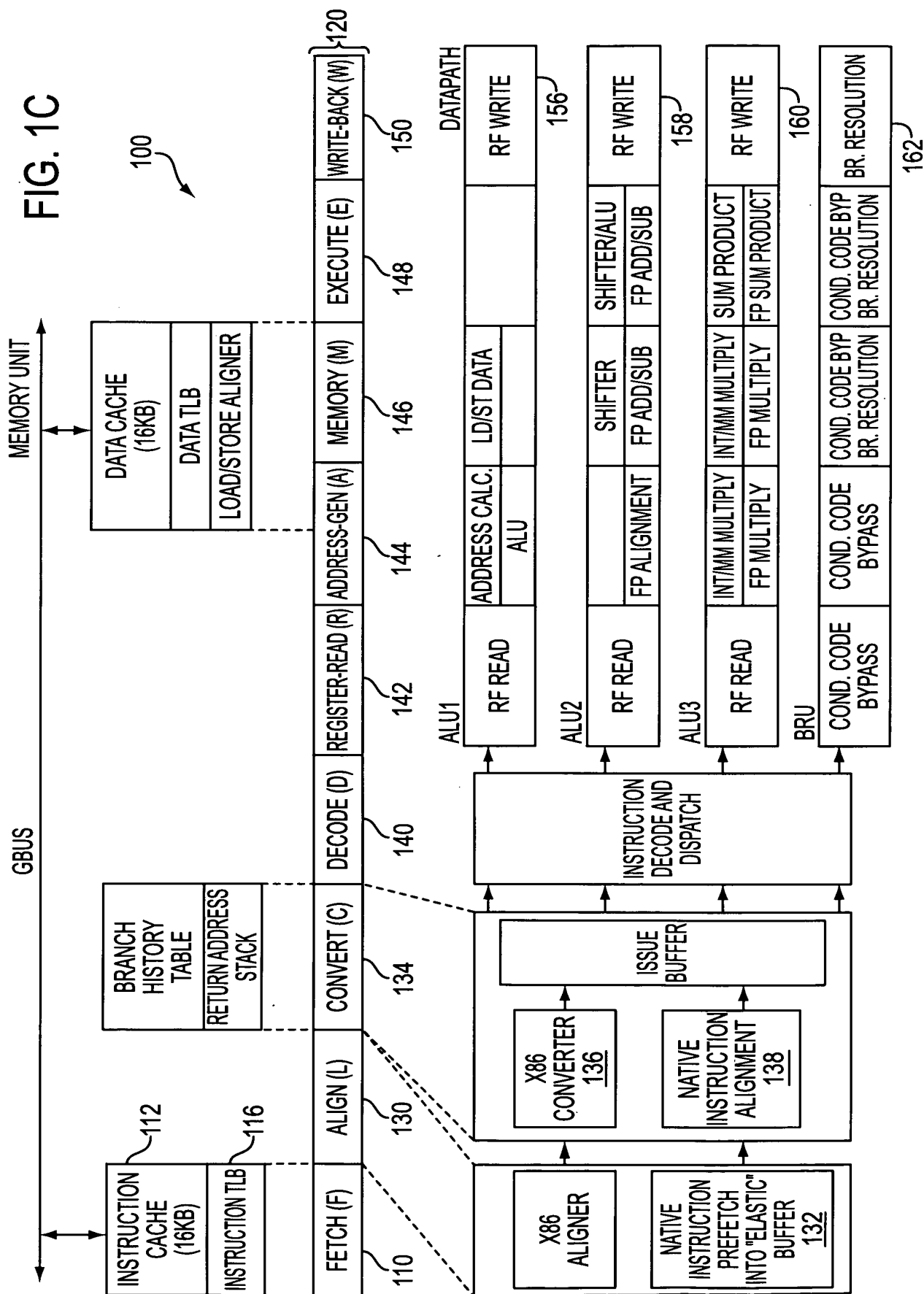


FIG. 1B

FIG. 1C



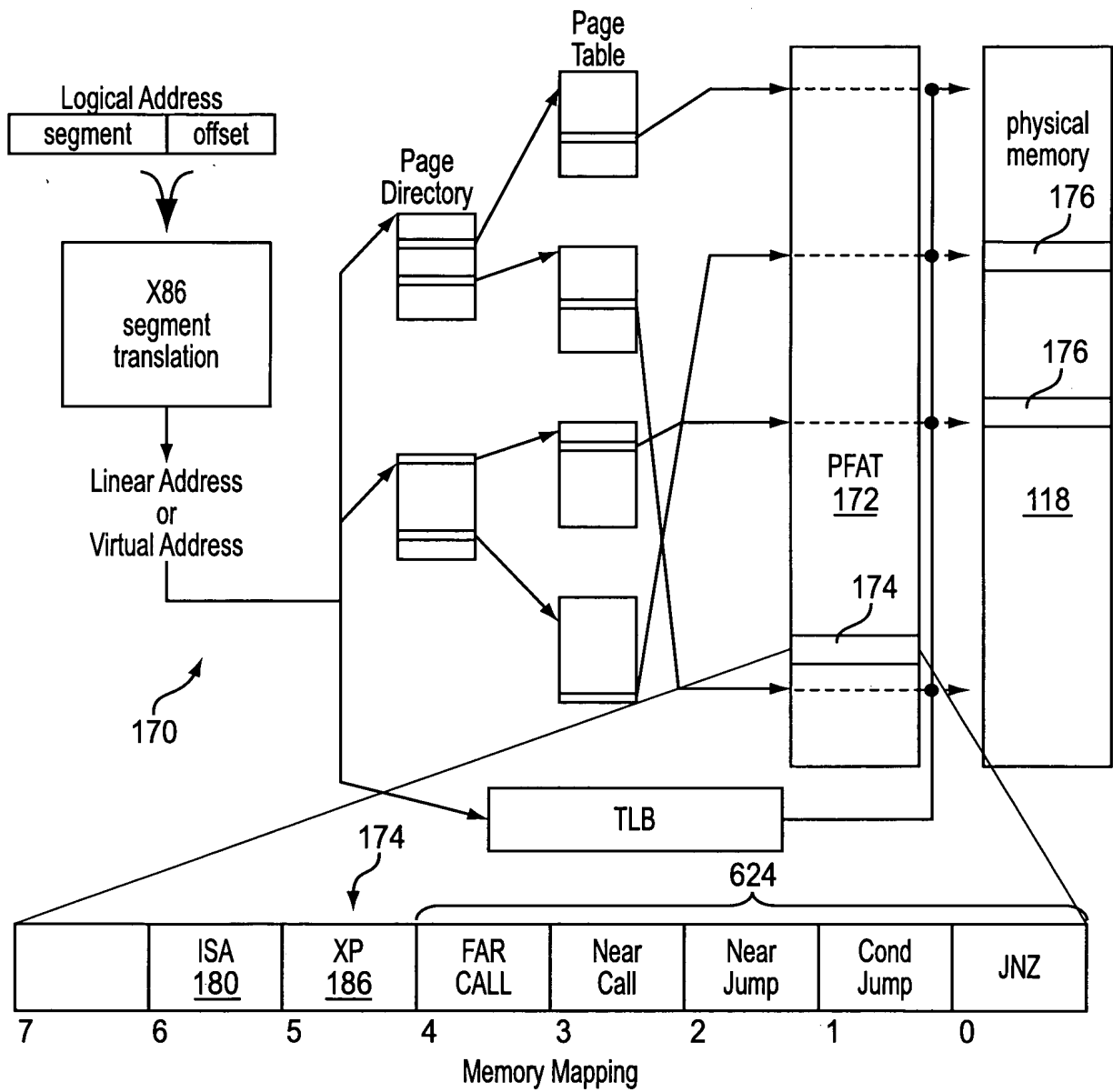
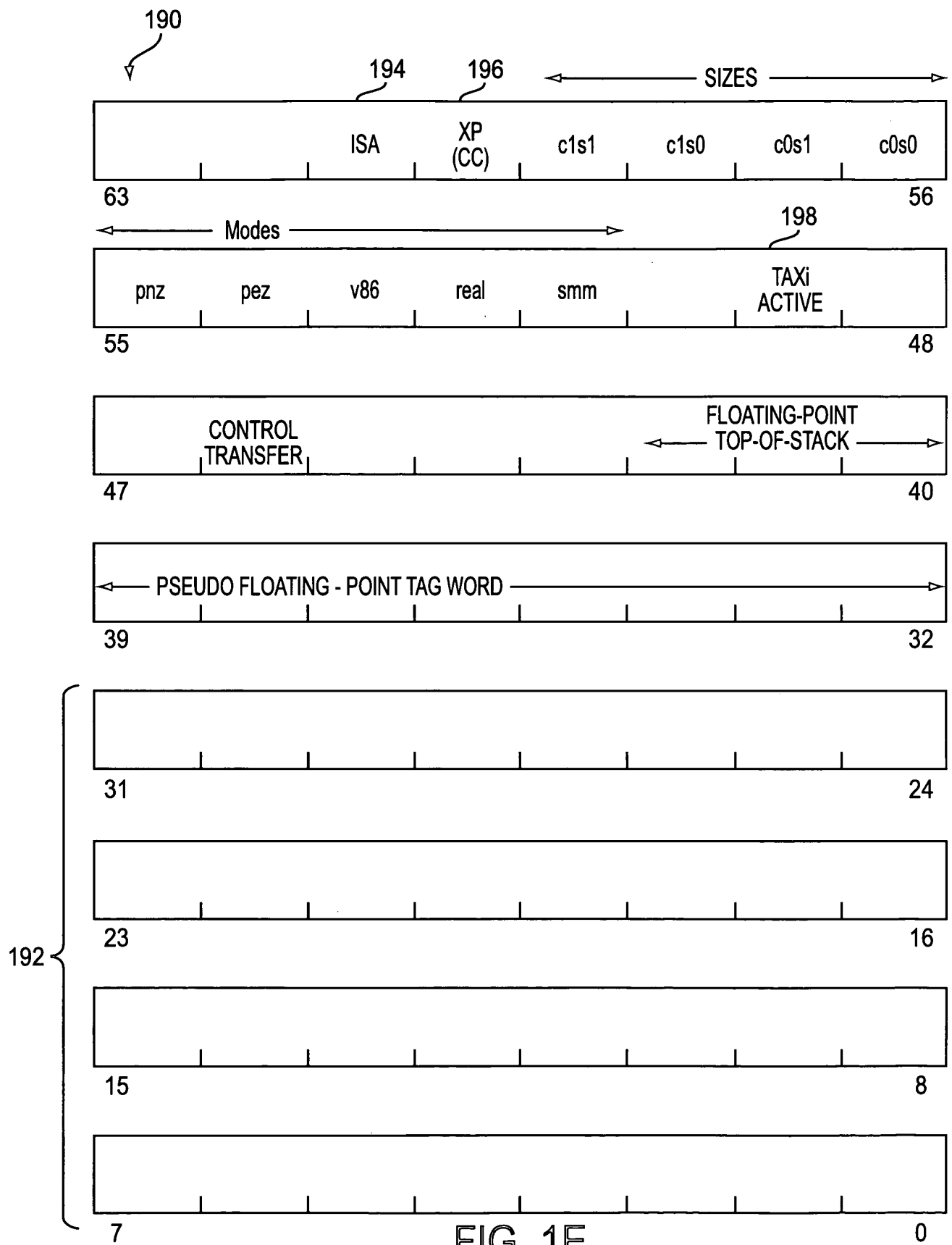


FIG. 1D



I-TLB PROPERTY BITS	DECODED PROPERTY VALUES			PROTECTED INTERPRETATION	INSTRUCTIONS SENT TO:	COLLECT PROFILE TRACE- PACKETS?	PROBE FOR TRANSLATED CODE	I/O MEMORY REFERENCE EXCEPTIONS
	ISA 194	CC 200						
00	TAP	TAP	NO	NATIVE CODE OBSERVING NATIVE RISC _y CALLING CONVENTIONS	NATIVE DECODER	NO	NO	FAULT IF SEG.tio
01	TAP	x86	NO	NATIVE CODE OBSERVING x86 CALLING CONVENTIONS	NATIVE DECODER	NO	NO	FAULT IF SEG.tio
10	x86	x86	NO	x86 CODE, UNPROTECTED - TAXI PROFILE COLLECTION ONLY	x86 HW CONVERTER	IF ENABLED	NO	TRAP IF PROFILING
11	x86	x86	YES	x86 CODE, PROTECTED - TAXI CODE MAY BE AVAILABLE	x86 HW CONVERTER	IF ENABLED	BASED ON I-TLB PROBE ATTRIBUTES	TRAP IF PROFILING

180,182,
184,186

184,186

FIG. 2A

TRANSITION (SOURCE => DEST) ISA & CC PROPERTY VALUES		HANDLER ACTION
00 => 00		NO TRANSITION EXCEPTION
00 => 01		VECT_xxx_X86_CC EXCEPTION - HANDLER CONVERTS FROM NATIVE TO x86 CONVENTIONS
00 => 1x		VECT_xxx_X86_CC EXCEPTION - HANDLER CONVERTS FROM NATIVE x86 CONVENTIONS, SETS UP EXPECTED EMULATOR AND PROFILING STATE
01 => 00		VECT_xxx_TAP_CC EXCEPTION - HANDLER CONVERTS FROM x86 TO NATIVE CONVENTIONS
01 => 01		NO TRANSITION EXCEPTION
01 => 1x		VECT_X86_ISA EXCEPTION [CONDITIONAL BASED ON PCW.X86_ISA_ENABLE FLAG] - SETS UP EXPECTED EMULATOR AND PROFILING STATE
1x => 00		VECT_xxx_TAP_CC EXCEPTION - HANDLER CONVERTS FROM x86 TO NATIVE CONVENTIONS
1x => 01		VECT_TAP_ISA EXCEPTION [CONDITIONAL BASED PCW.TAP_ISA_ENABLE FLAG] - NO CONVENTION CONVERSION NECESSARY
1x => 10		NO TRANSITION EXCEPTION - [PROFILE COMPLETE POSSIBLE, PROBE POSSIBLE]
1x => 11		NO TRANSITION EXCEPTION - [PROFILE COMPLETE POSSIBLE, PROBE NOT POSSIBLE]

FIG. 2B

NAME	DESCRIPTION	TYPE
VECT_call_X86_CC	PUSH ARGS, RETURN ADDRESS, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
VECT_jump_X86_CC	SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
VECT_ret_no_fp_X86_CC	RETURN VALUE TO EAX:EDX, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
VECT_ret_fp_X86_CC	RETURN VALUE TO x86 FP STACK, SET UP x86 STATE	FAULT ON TARGET INSTRUCTION
VECT_call_TAP_CC	x86 STACK ARGS, RETURN ADDRESS TO REGISTERS	FAULT ON TARGET INSTRUCTION
VECT_jump_TAP_CC	x86 STACK ARGS TO REGISTERS	FAULT ON TARGET INSTRUCTION
VECT_ret_no_fp_TAP_CC	RETURN VALUE TO RV0	FAULT ON TARGET INSTRUCTION
VECT_ret_any_TAP_CC	RETURN TYPE UNKNOWN, SETUP RV0 AND RVDP	FAULT ON TARGET INSTRUCTION

FIG. 2C

FLAT 32-BIT "NEAR" ADDRESS SPACE

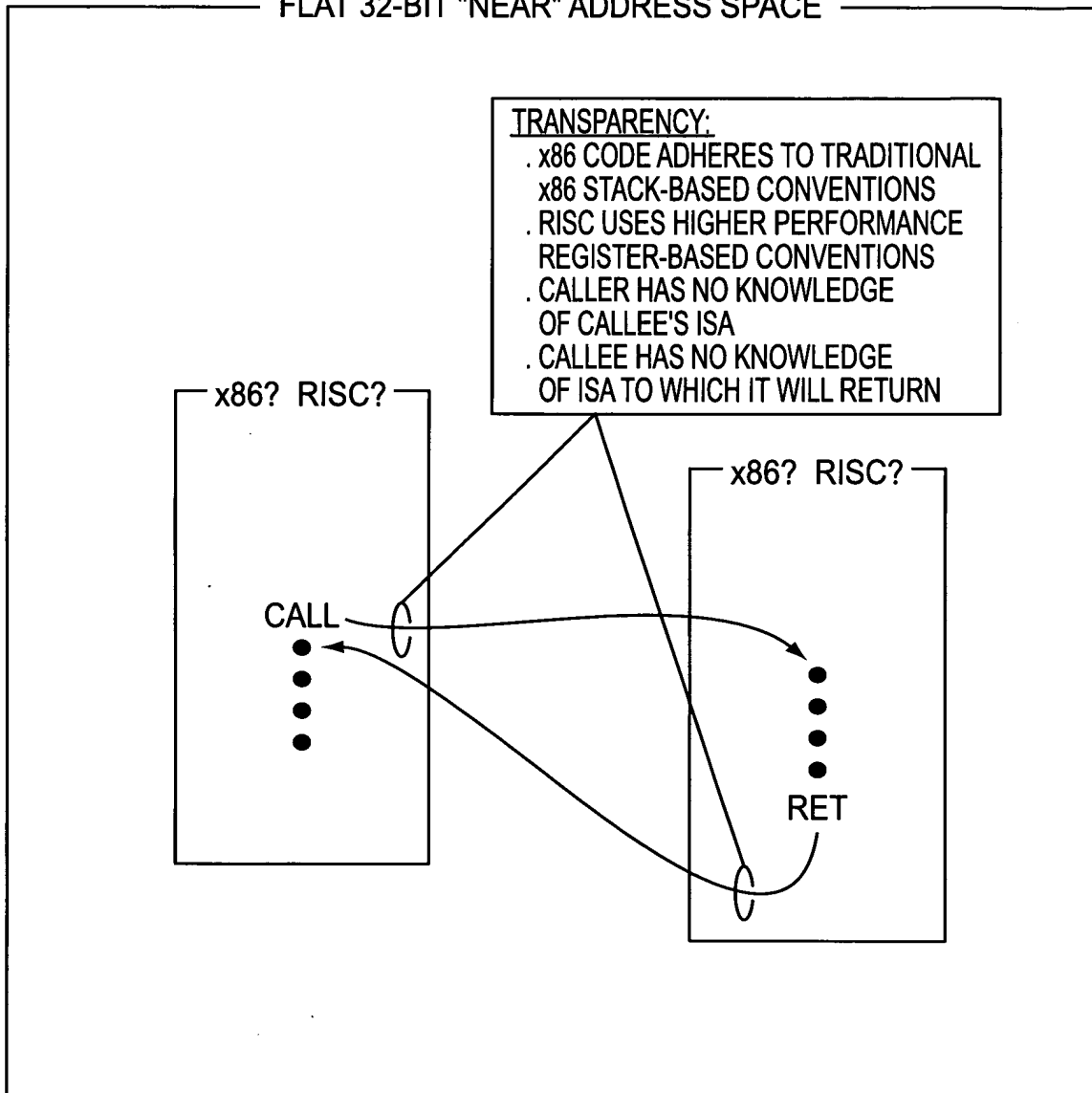


FIG. 3B

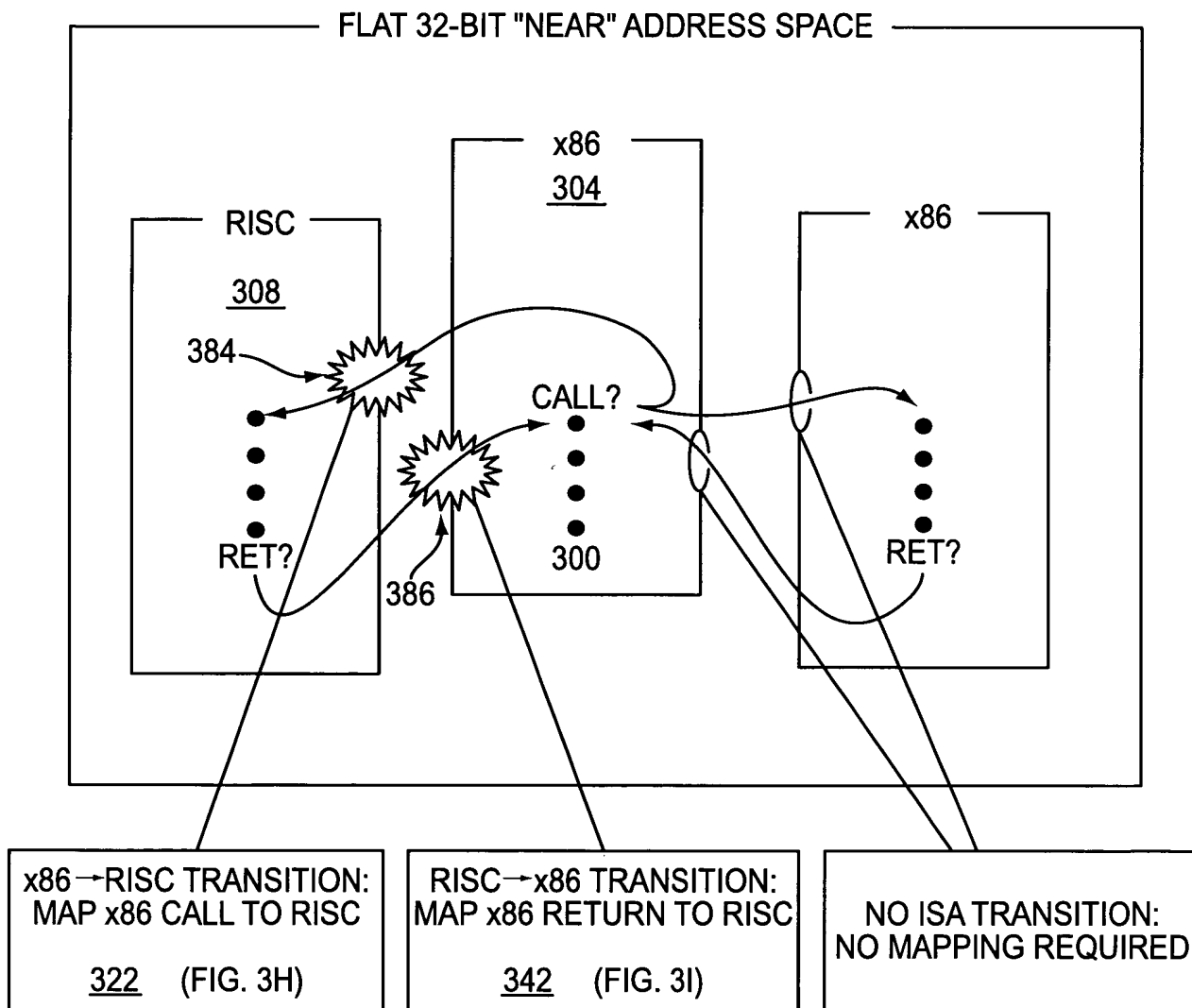


FIG. 3C

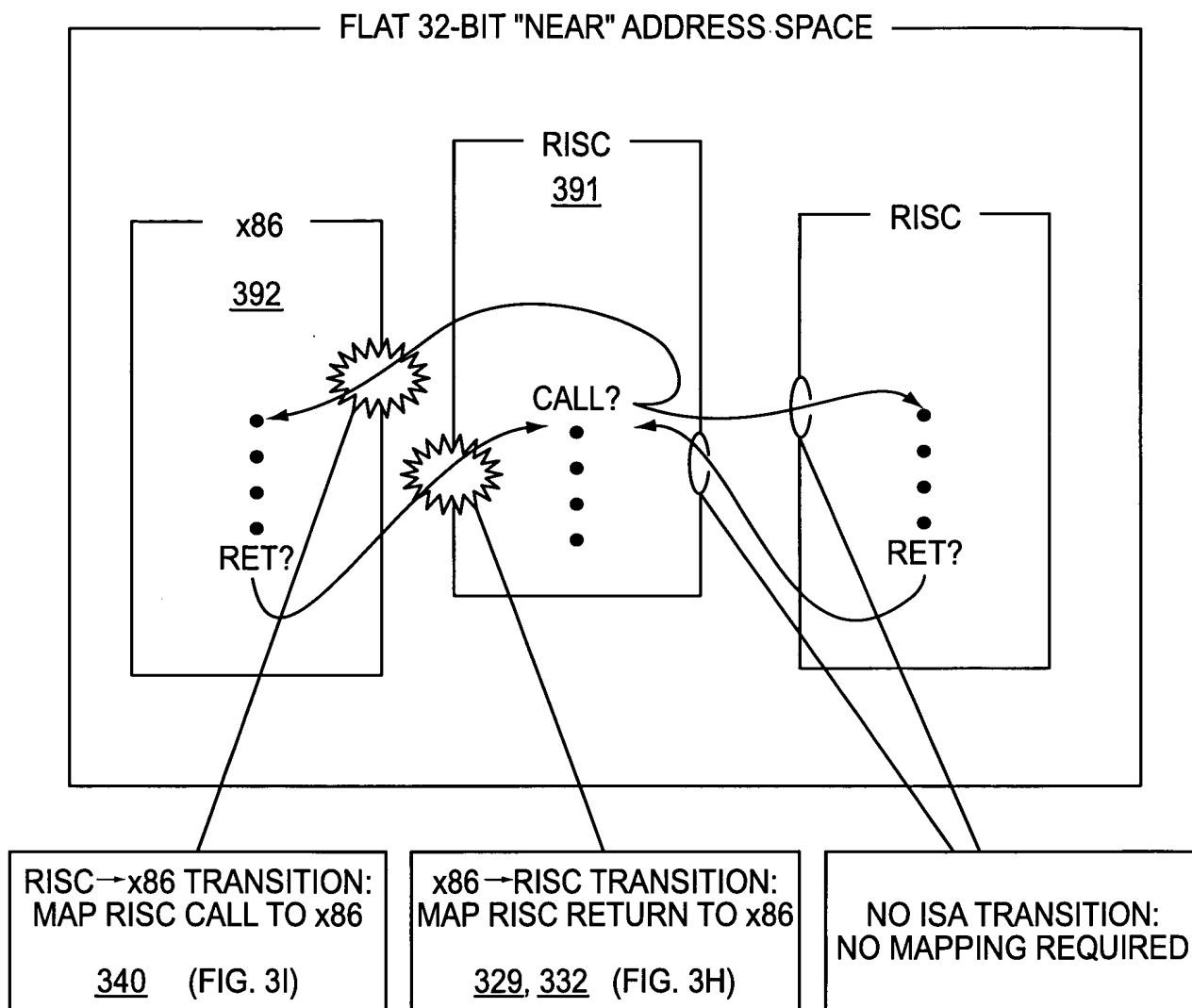


FIG. 3D

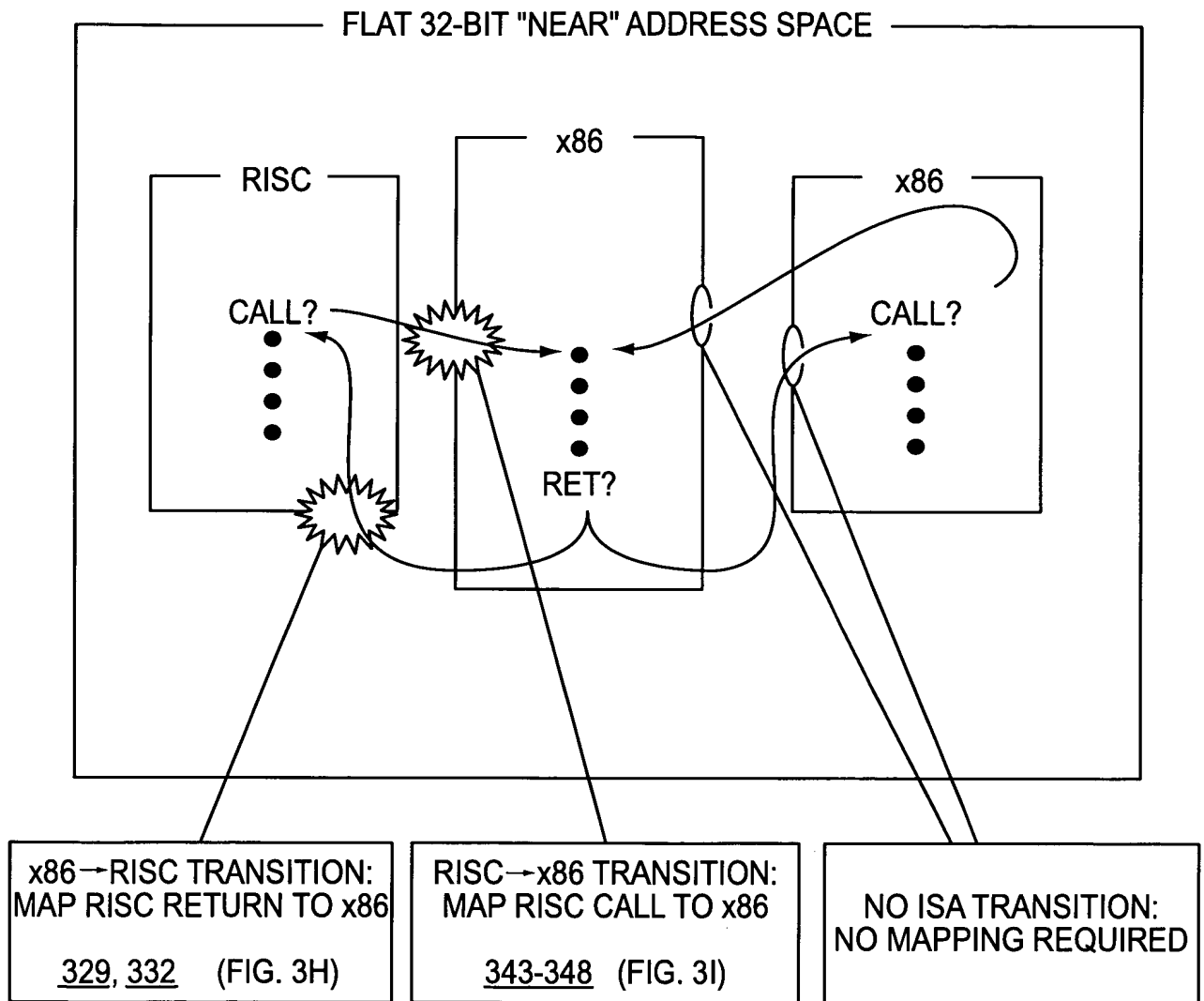


FIG. 3E

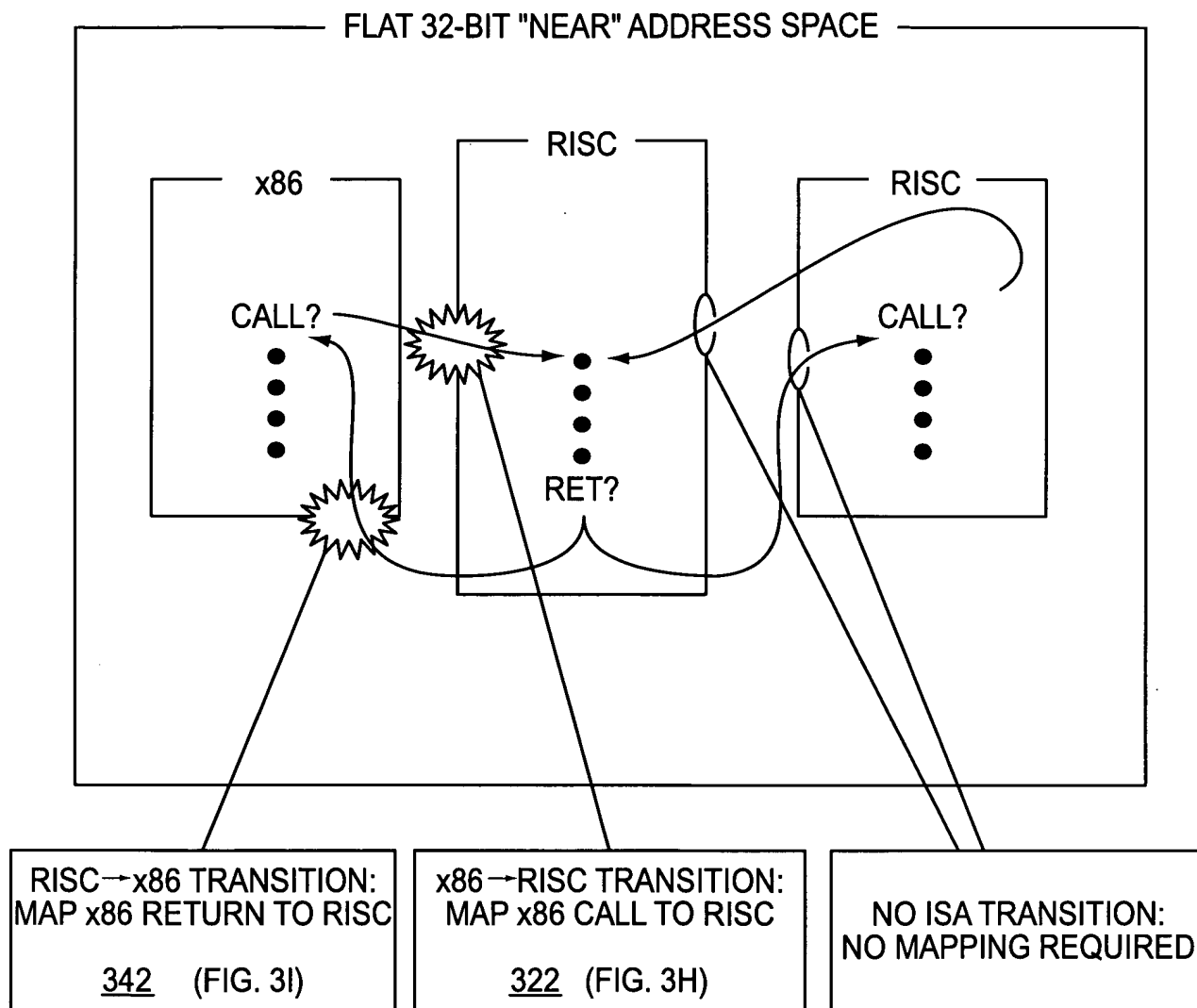


FIG. 3F

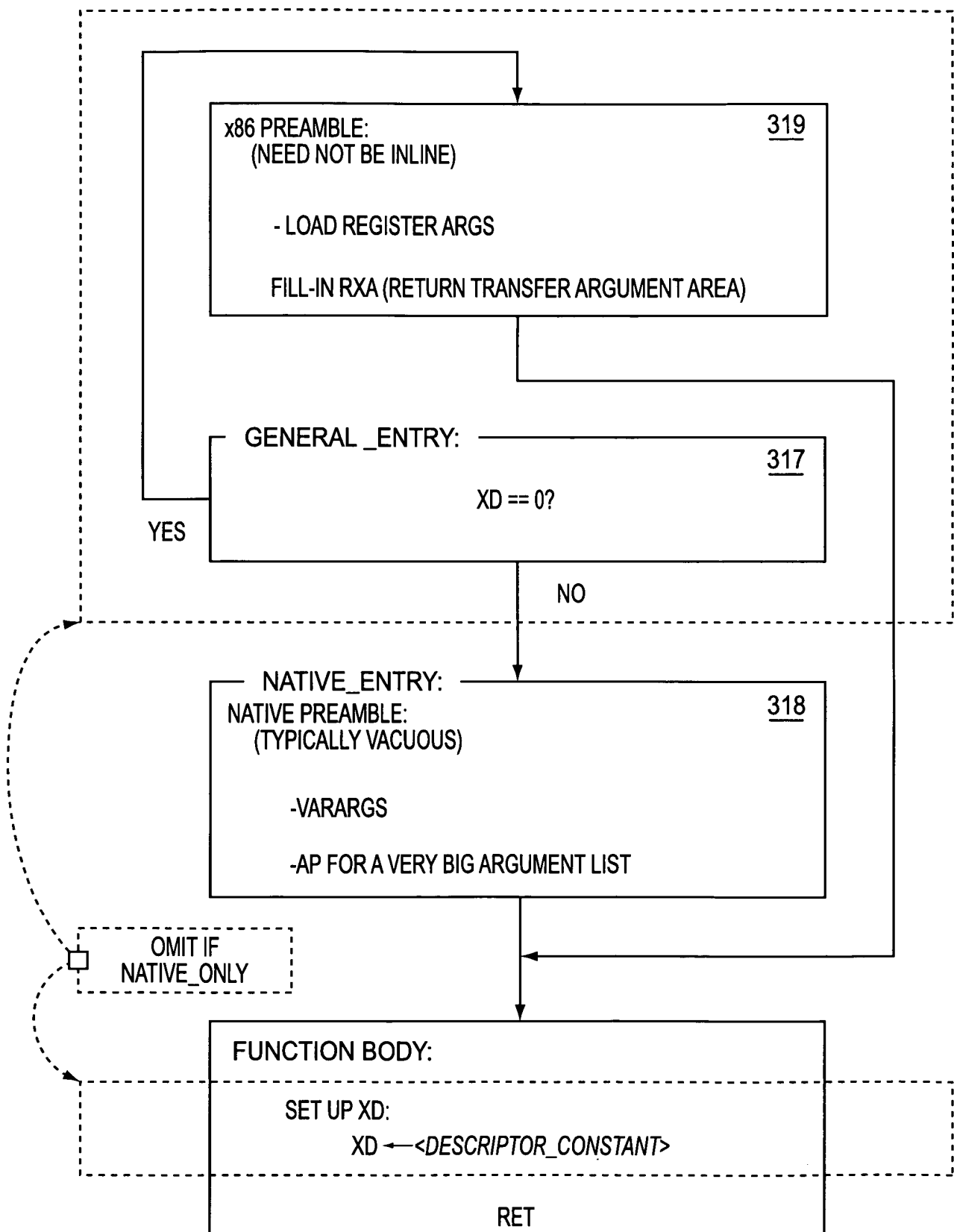


FIG. 3G

X86-to Tapestry transition exception handler

```

// This handler is entered under the following conditions:
// 1. An x86 caller invokes a native function
// 2. An x86 function returns to a native caller
// 3. x86 software returns to or resumes an interrupted native function following
//    an external asynchronous interrupt, a processor exception, or a context switch
dispatch on the two least-significant bits of the destination address {
case "00" // calling a native subprogram
    // copy linkage and stack frame information and call parameters from the memory
    // stack to the analogous Tapestry registers
    LR ← [SP++] // set up linkage register 323
    AP ← SP // address of first argument 324
    SP ← SP - 8 // allocate return transfer argument area 326
    SP ← SP & (-32) // round the stack pointer down to a 0 mod 32 boundary 327
    XD ← 0 // inform callee that caller uses X86 calling conventions 328
case "01" // resuming an X86 thread suspended during execution of a native routine
    if the redundant copies of the save slot number in EAX and EDX do not match or if
        the redundant copies of the timestamp in EBX:ECX and ESI:EDI do not match {
        // some form of bug or thread corruption has been detected
        goto TAPESTRY_CRASH_SYSTEM( thread-corruption-error-code ) 372
    }
    save the EBX:ECX timestamp in a 64-bit exception handler temporary register 373
    (this will not be overwritten during restoration of the full native context)
    use save slot number in EAX to locate actual save slot storage 374
    restore full entire native context (includes new values for all x86 registers) 375
    if save slot's timestamp does not match the saved timestamp { 376
        // save slot has been reallocated; save slot exhaustion has been detected
        goto TAPESTRY_CRASH_SYSTEM( save-slot-overwritten-error-code ) 377
    }
    free the save slot 378
case "10" // returning from X86 callee to native caller, result already in registers
    RV0<63:32> ← edx<31:00> // in case result is 64 bits 333
    convert the FP top-of-stack value from 80 bit X86 form to 64-bit form in RVDP 334
    SP ← ESI // restore SP from time of call 337
case "11" // returning from X86 callee to native caller, load large result from memory
    RV0..RV3 ← load 32 bytes from [ESI-32] // (guaranteed naturally aligned) 330
    SP ← ESI // restore SP from time of call 337
}
EPC ← EPC & -4 // reset the two low-order bits to zero 336
RFE 338

```

FIG. 3H

340

Tapestry-to-X86 transition exception handler

// This handler is entered under the following conditions:

// 1. a native caller invokes an x86 function

// 2. a native function returns to an x86 caller

switch on XD<3:0> {

341

XD_RET_FP: // result type is floating point

FO/FI ← FINFLATE.de(RVDP) // X86 FP results are 80 bits

SP ← from RXA save // discard RXA, pad, args

FPCW ← image after FINIT & push // FP stack has 1 entry

goto EXIT

XD_RET_WRITEBACK: // store result to @RVA, leave RVA in eax

RVA ← from RXA save // address of result area

copy decode(XD<8:4>) bytes from RV0..RV3 to [RVA]

eax ← RVA // X86 expects RVA in eax

SP ← from RXA save // discard RXA, pad, args

FPCW ← image after FINIT // FP stack is empty

goto EXIT

XD_RET_SCALAR: // result in eax:edx

edx<31:00> ← eax<63:32> // in case result is 64 bits

SP ← from RXA save // discard RXA, pad, args

FPCW ← image after FINIT // FP stack is empty

goto EXIT

XD_CALL_HIDDEN_TEMP: // allocate 32 byte aligned hidden temp

esi ← SP // stack cut back on return

SP ← SP - 32 // allocate max size temp

RVA ← SP // RVA consumed later by RR

LR<1:0> ← "11" // flag address for return & reload

goto CALL_COMMON

default: // remaining XD_CALL_xxx encodings

esi ← SP // stack cut back on return

LR<1:0> ← "10" // flag address for return

CALL_COMMON: 347

interpret XD to push and/or reposition args

[--SP] ← LR // push LR as return address

EXIT: 348

setup emulator context and profiling ring buffer pointer

} RFE 349

// to original target

}

FIG. 31

interrupt/exception handler of Tapestry operating system:

// Control vectors here when a synchronous exception or asynchronous interrupt is to be
// exported to / manifested in an x86 machine.

// The interrupt is directed to something within the virtual X86, and thus there is a possibility
// that the X86 operating system will context switch. So we need to distinguish two cases:
// either the running process has only X86 state that is relevant to save, or
// there is extended state that must be saved and associated with the current machine context
// (e.g., extended state in a Tapestry library call in behalf of a process managed by X86 OS)
if execution was interrupted in the converter - EPC.ISA == X86 {

// no dependence on extended/native state possible, hence no need to save any } 351
goto EM86_Deliver_Interrupt(interrupt-byte)

} else if EPC.Taxi_Active {

// A Taxi translated version of some X86 code was running. Taxi will rollback to an
// x86 instruction boundary. Then, if the rollback was induced by an asynchronous external
// interrupt, Taxi will deliver the appropriate x86 interrupt. Else, the rollback was induced
// by a synchronous event so Taxi will resume execution in the converter, retriggering the } 353
// exception but this time with EPC.ISA == X86
goto TAXi_Rollback(asynchronous-flag, interrupt-byte)

} else if EPC.EM86 {

// The emulator has been interrupted. The emulator is coded to allow for such
// conditions and permits re-entry during long running routines (e.g. far call through a gate)
// to deliver external interrupts } 354
goto EM86_Deliver_Interrupt(interrupt-byte)

} else {

// This is the most difficult case - the machine was executing native Tapestry code on
// behalf of an X86 thread. The X86 operating system may context switch. We must save
// all native state and be able to locate it again when the x86 thread is resumed.

361
allocate a free save slot; if unavailable free the save slot with oldest timestamp and try again
save the entire native state (both the X86 and the extended state) } 362

save the X86 EIP in the save slot

overwrite the two low-order bits of EPC with "01" (will become X86 interrupt EIP) } 363

store the 64-bit timestamp in the save slot, in the X86 EBX:ECX register pair (and,
for further security, store a redundant copy in the X86 ESI:EDI register pair) } 364

store the a number of the allocated save slot in the X86 EAX register (and, again for
further security, store a redundant copy in the X86 EDX register) } 365

goto EM86_Deliver_Interrupt(interrupt-byte) } 369

}

FIG. 3J

```

typedef struct {
    save_slot_t * newer, // pointer to next-most-recently-allocated save slot } 379c
    save_slot_t * older; // pointer to next-older save slot
    unsigned int64 epc; // saved exception PC/IP
    unsigned int64 pcw; // saved exception PCW (program control word)
    unsigned int64 registers[63]; // save the 63 writeable general registers } 356 } 355
    ... // other words of Tapestry context
    timestamp_t timestamp; // timestamp to detect buffer overrun } 358
    int save_slot_ID; // ID number of the save slot } 357
    boolean save_slot_is_full; // full / empty flag } 359
} save_slot_t;

save_slot_t * save_slot_head; // pointer to the head of the queue } 379a
save_slot_t * save_slot_tail; // pointer to the tail of the queue } 379b

```

system initialization
 reserve several pages of unpagged memory for save slots

FIG. 3K

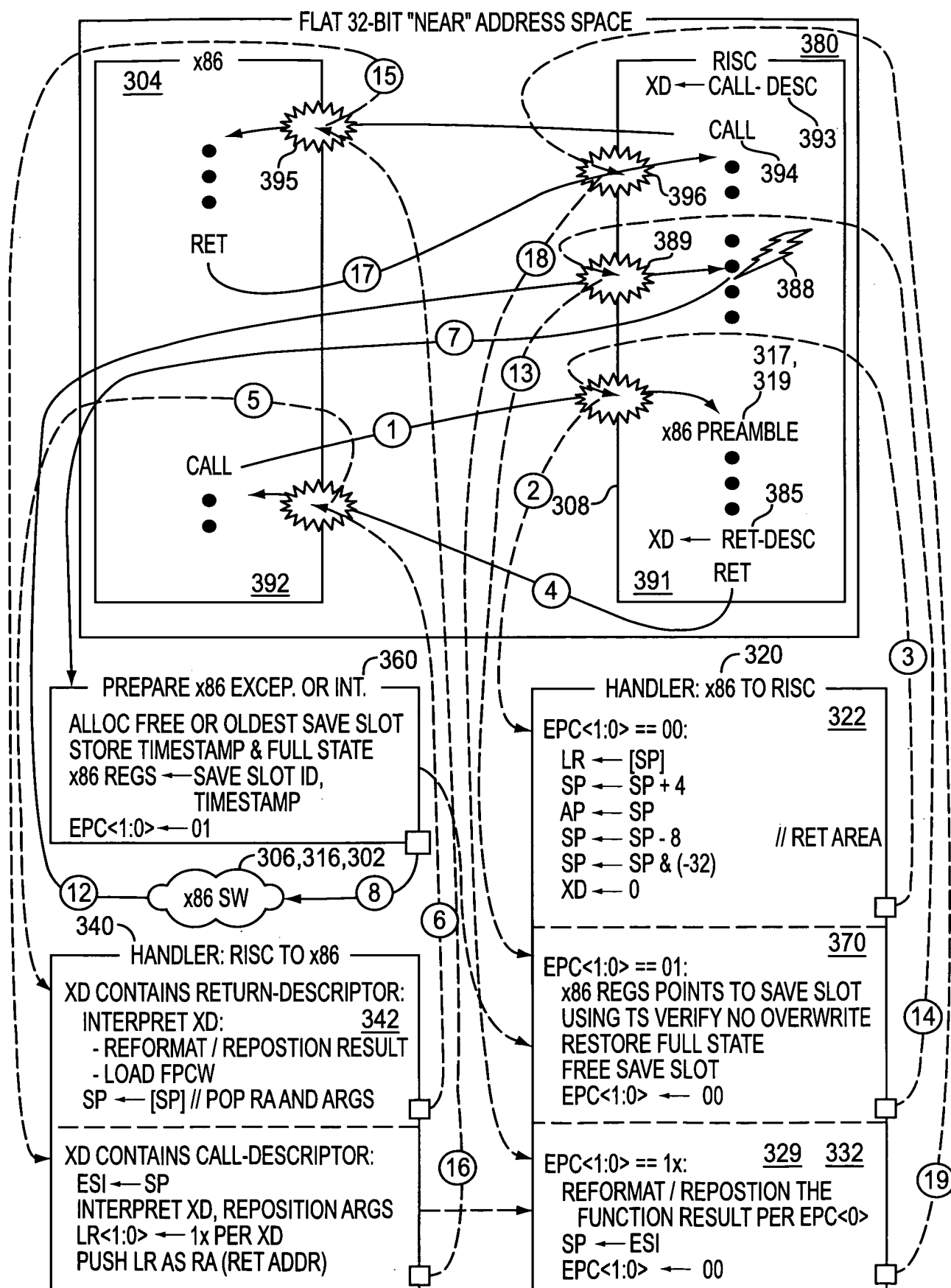


FIG. 3L

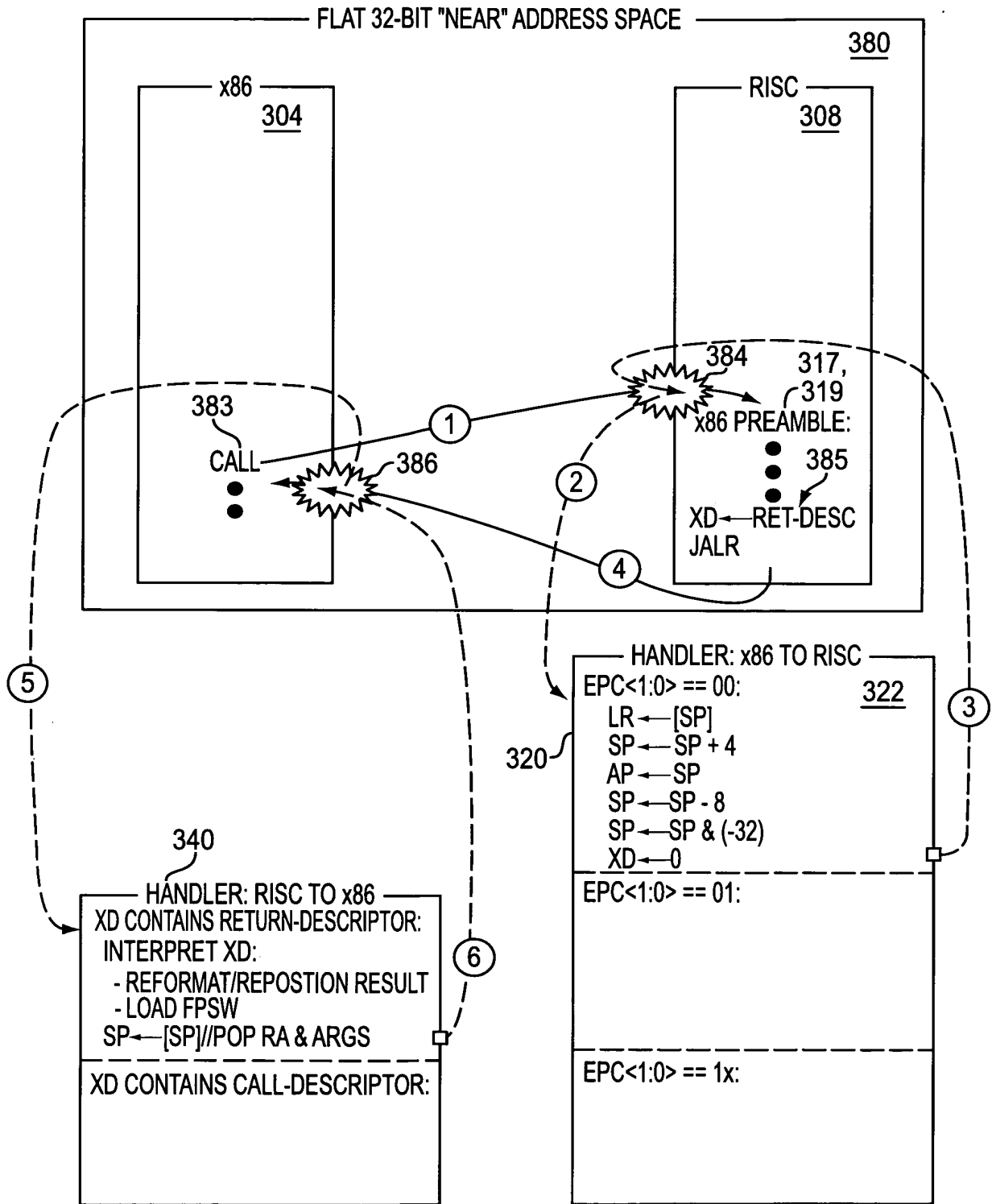


FIG. 3M

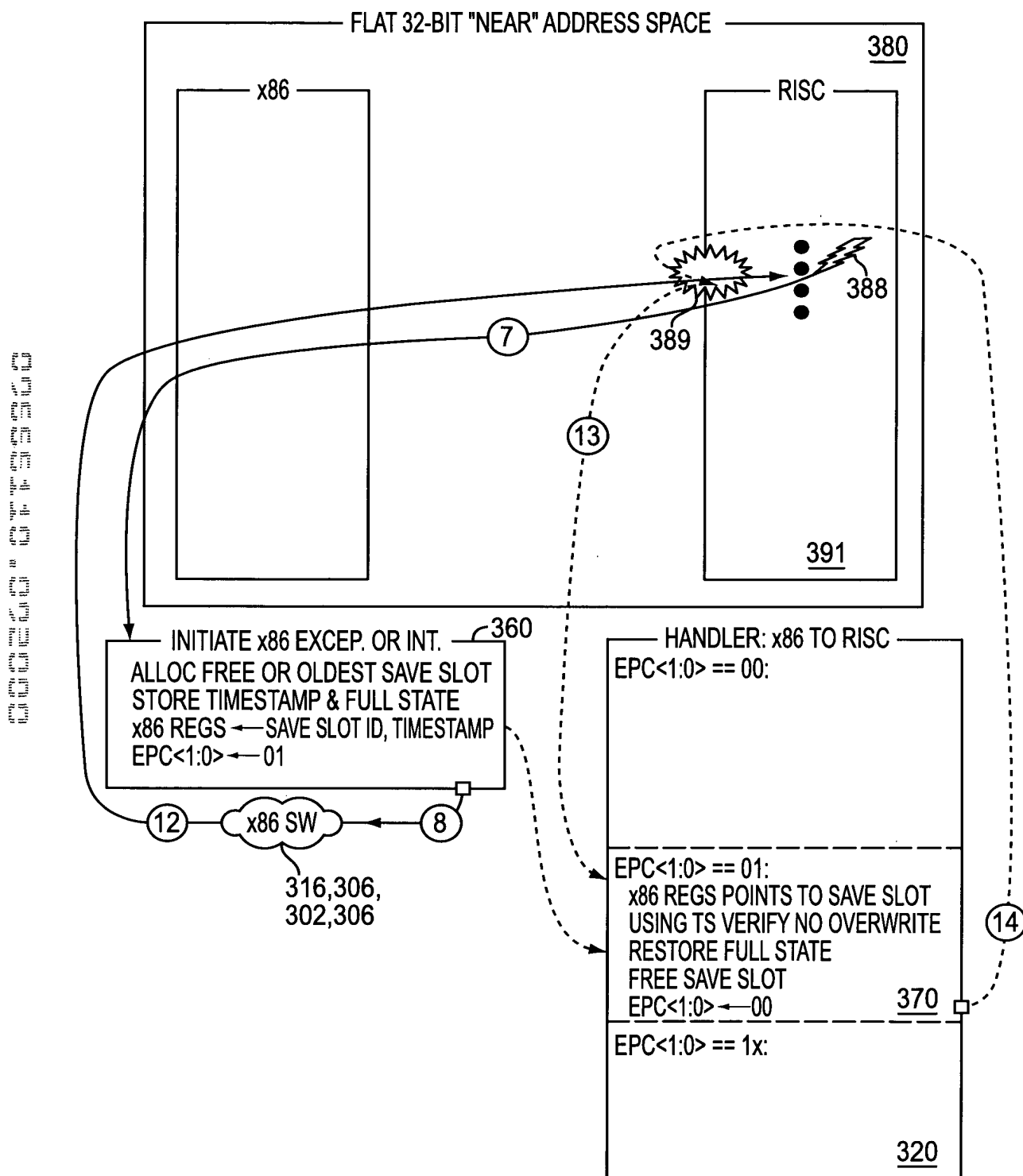


FIG. 3N

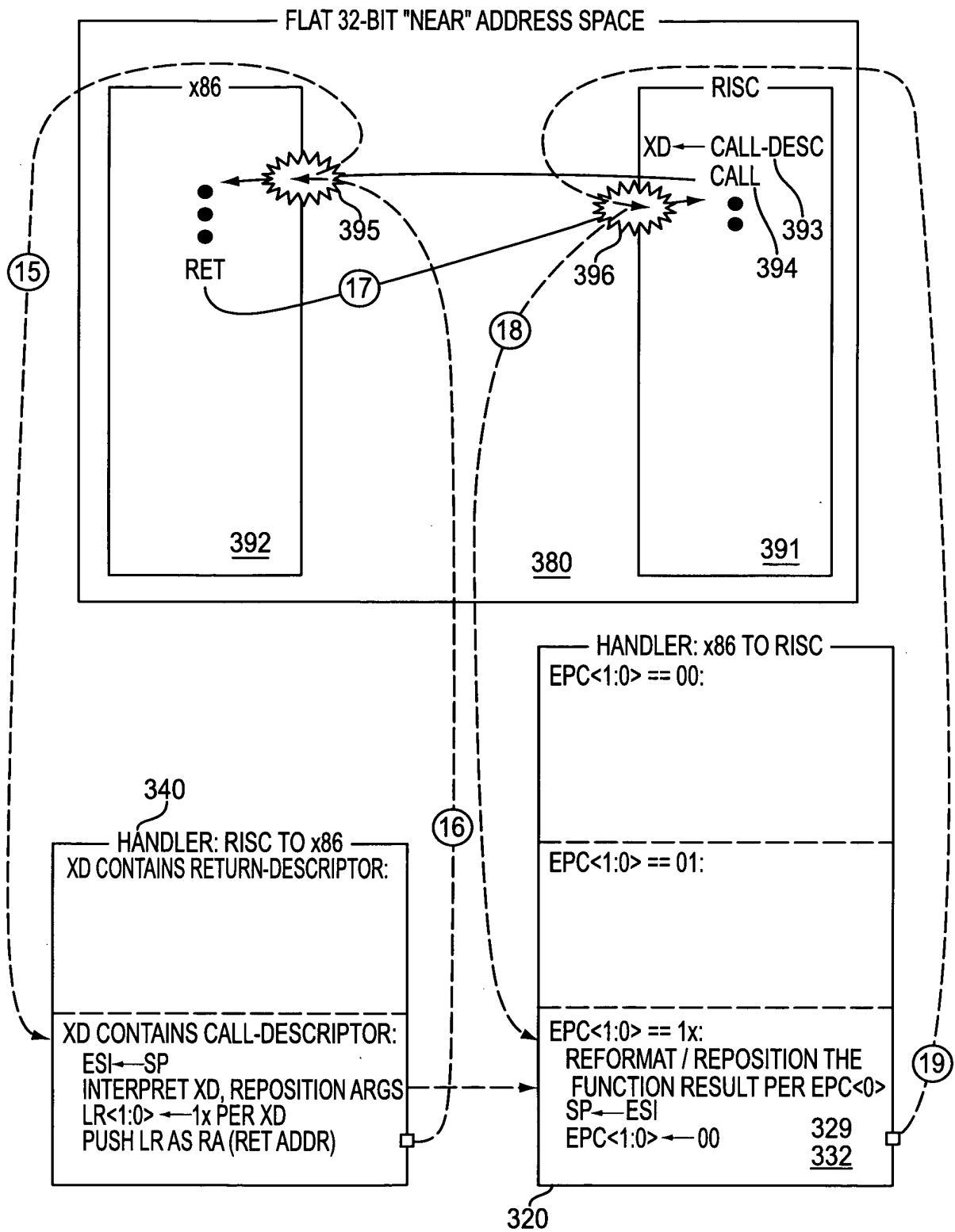
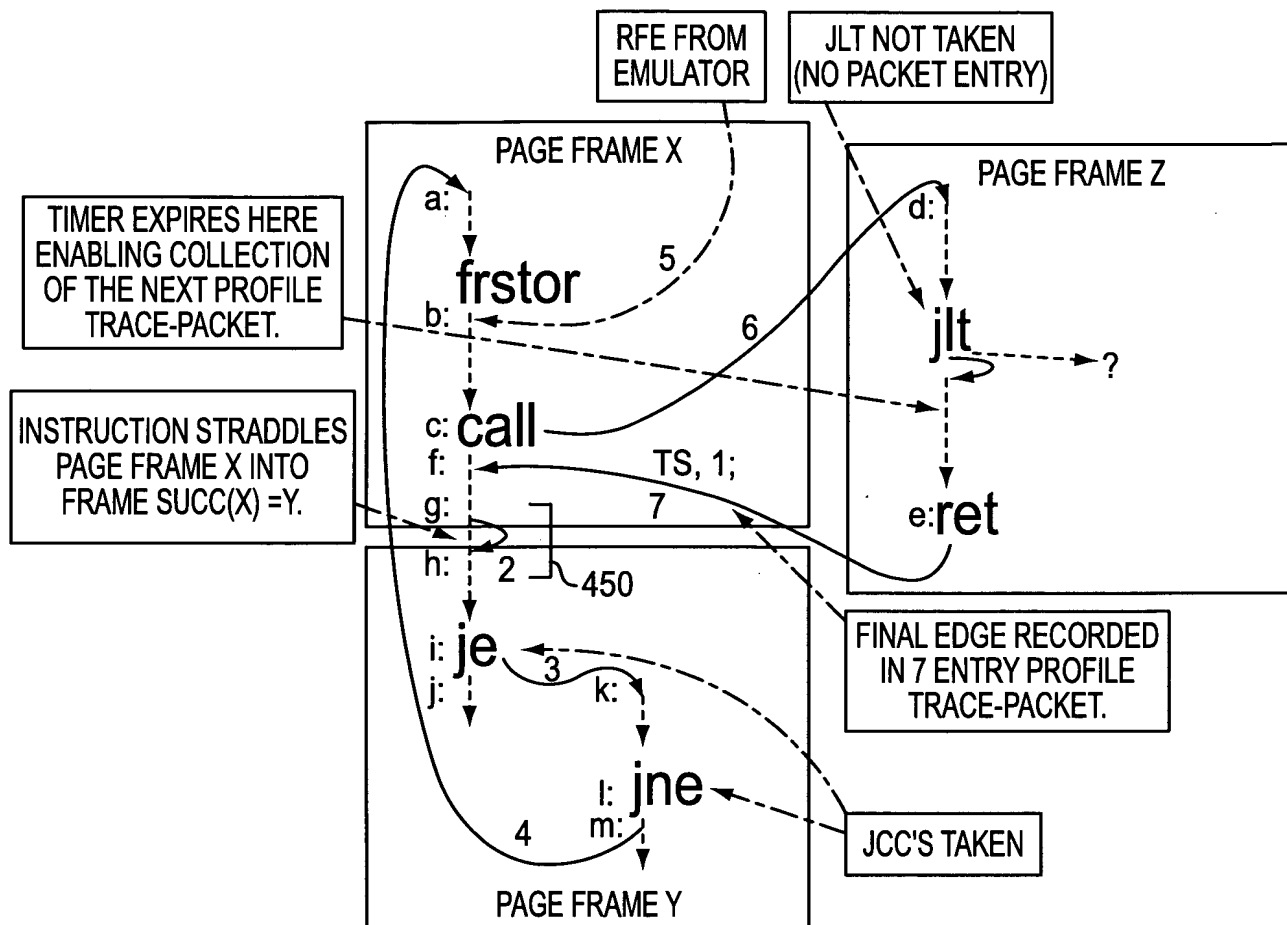


FIG. 30



7 ENTRY TRACE PACKET

ENTRY	EVENT CODE	DONE ADDR	NEXT ADDR
64 BIT TIME STAMP			
1	RET	x86 CONTEXT	<i>phys X:f</i>
2	NEW PAGE	<i>phys Y:g</i>	<i>phys Y:h</i>
3	JCC FORWARD	<i>phys Y:i</i>	<i>phys Y:k</i>
4	JNZ BACKWARD	<i>phys Y:l</i>	<i>phys X:a</i>
5	SEQ; ENV CHANGE	x86 CONTEXT	<i>phys X:b</i>
6	IP-REL NEAR CALL	<i>phys X:c</i>	<i>phys Z:d</i>
7	NEAR RET	<i>phys Z:e</i>	<i>phys X:f</i>

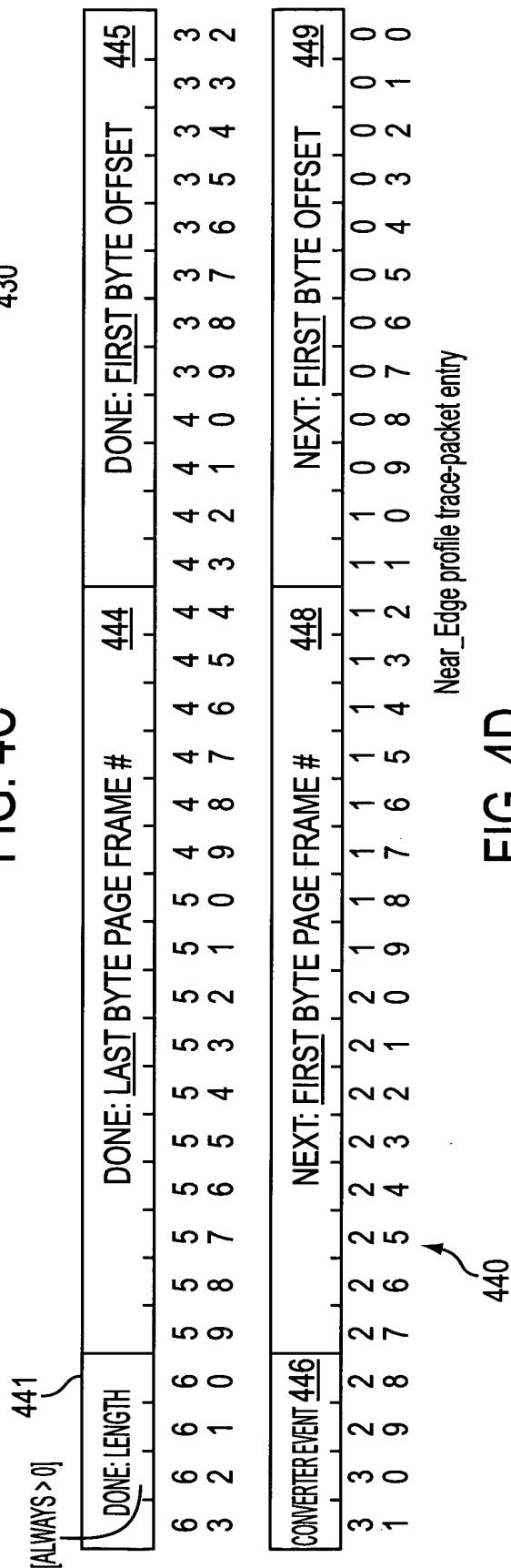
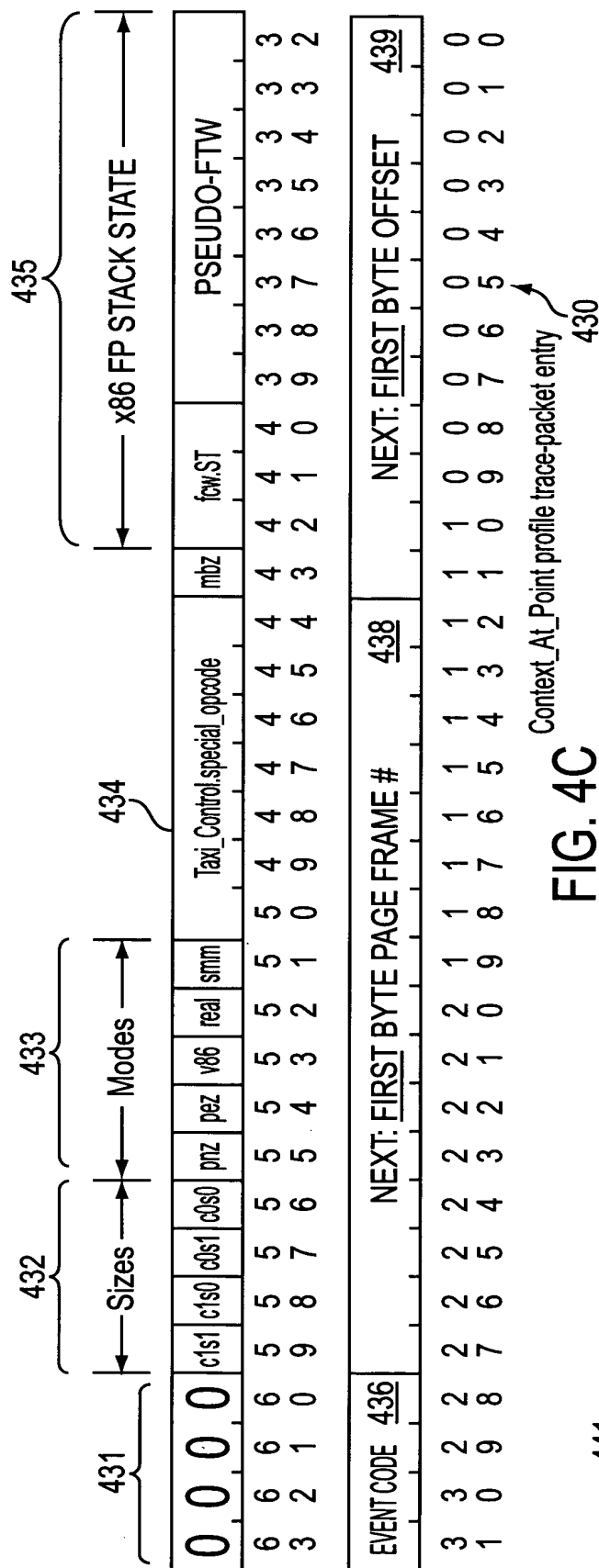
420 {

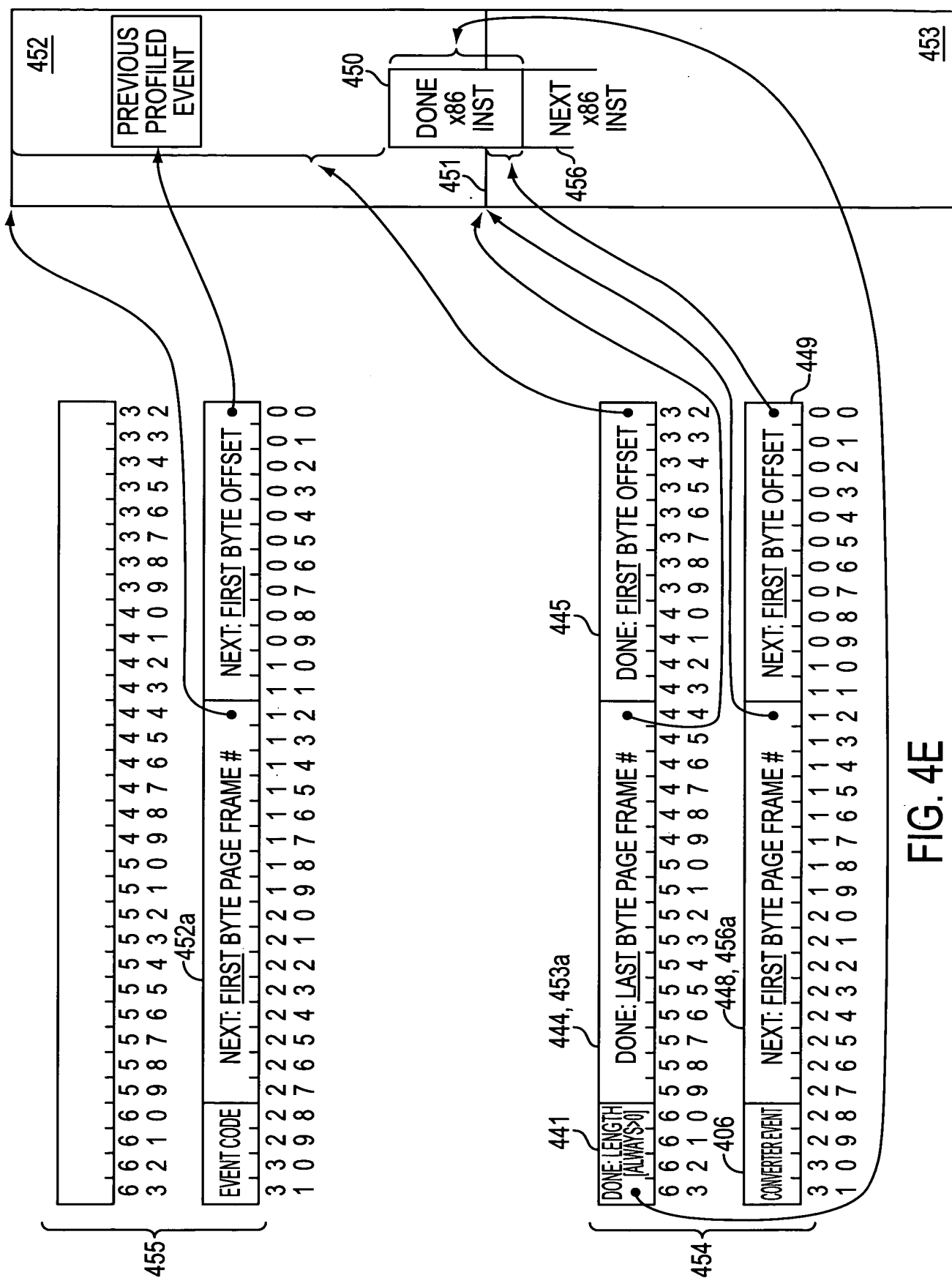
430
440, 454
440
440
430
440
440

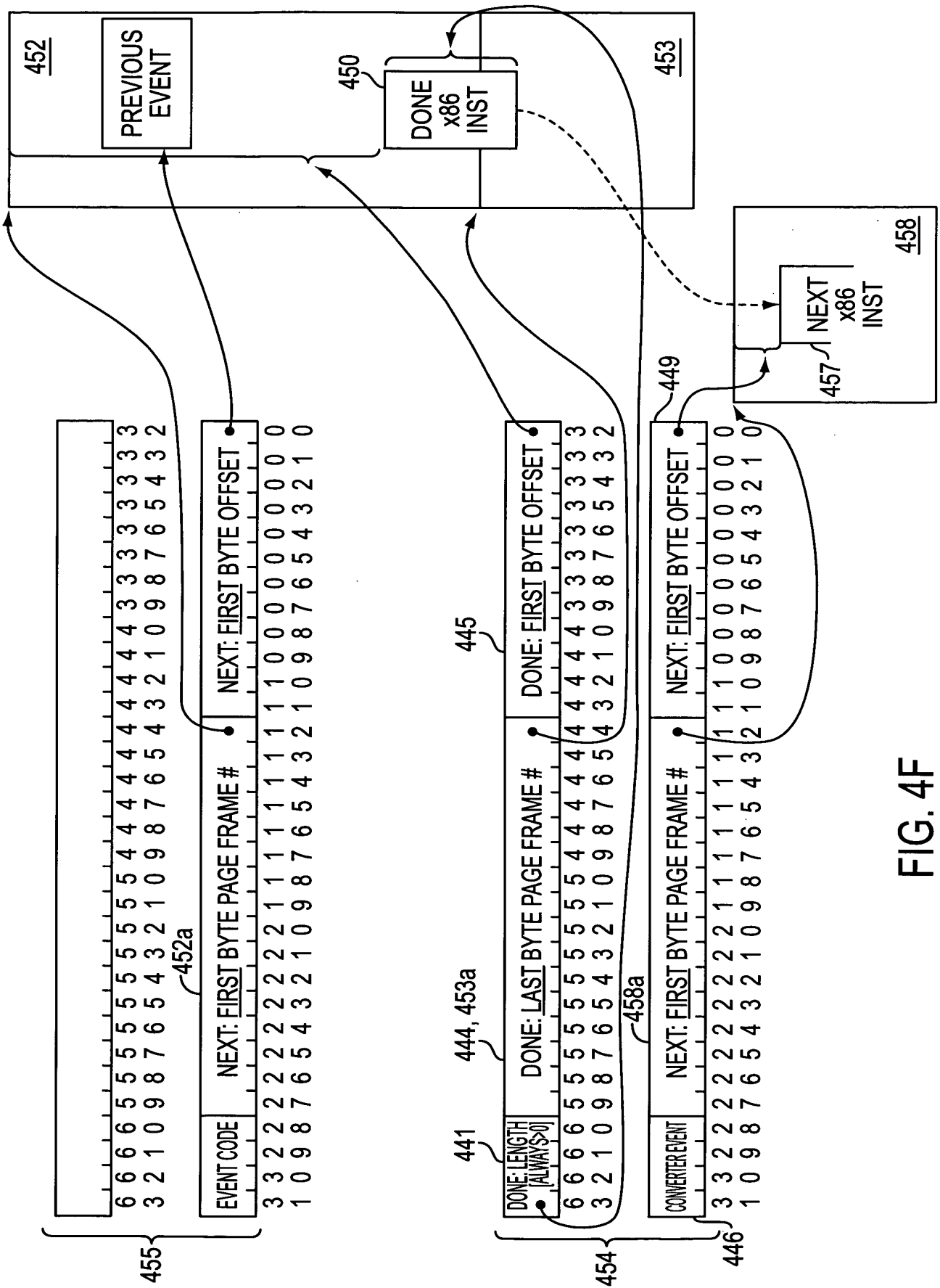
FIG. 4A

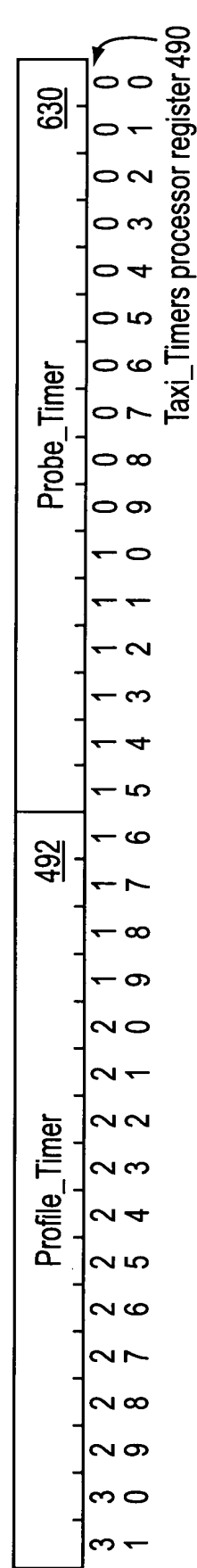
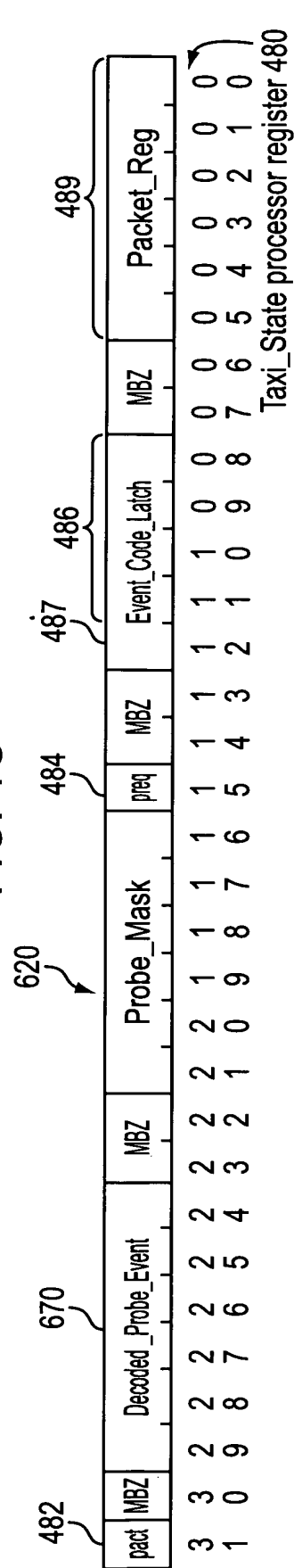
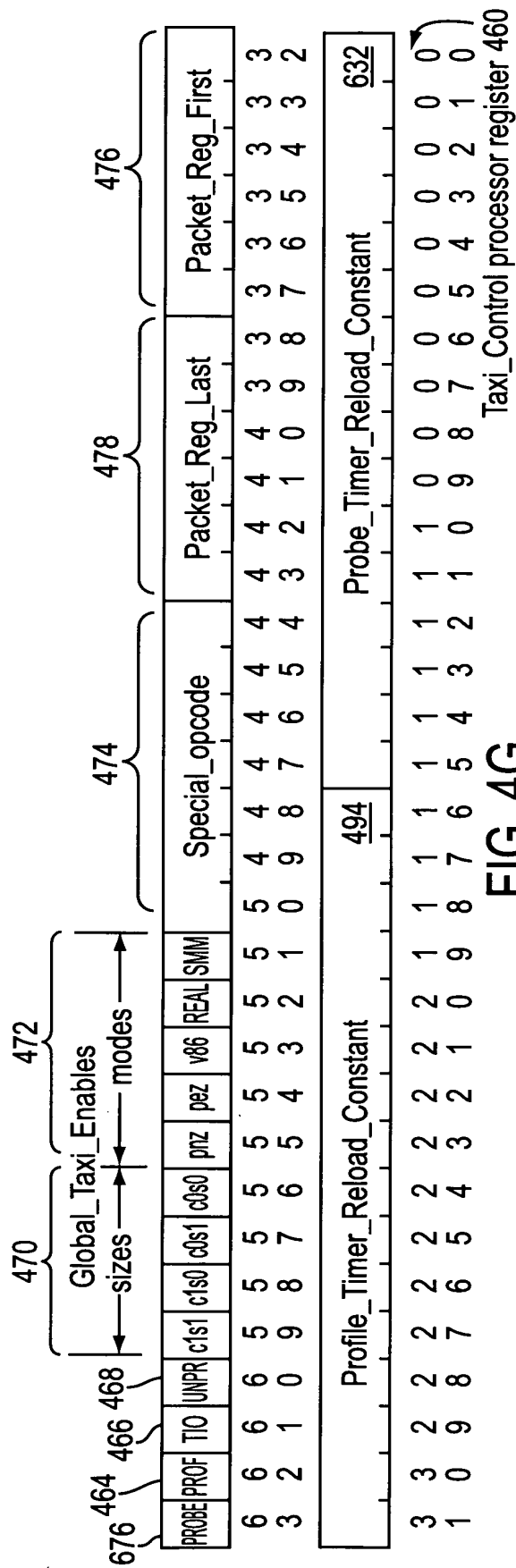
SOURCE		414		416	418	610	612
CODE 402	EVENT	REUSE EVENT CODE					PROBE EVENT BIT- TLB PROBE ATTRIBUTE OR EMULATOR PROBE
412 RFE (CONTEXT AT POINT ENTRY)	0.0000	DEFAULT (x86 TRANSPARENT) EVENT, REUSE ALL CONVERTER VALUES	YES		NO		REUSE EVENT CODE
	0.0001	SIMPLE x86 INSTRUCTION COMPLETION (REUSE EVENT CODE)	YES		NO		REUSE EVENT CODE
	0.0010	PROBE EXCEPTION FAILED	YES		NO		REUSE EVENT CODE
	0.0011	PROBE EXCEPTION FAILED, RELOAD PROBE TIMER	YES		NO		REUSE EVENT CODE
	0.0100	FLUSH EVENT	NO	NO	NO	NO	-
	0.0101	SEQUENTIAL; EXECUTION ENVIRONMENT CHANGED - FORCE EVENT	NO	YES	NO	NO	-
	0.0110	FAR RET	NO	YES	YES	NO	-
	0.0111	IRET	NO	YES	NO	NO	-
	0.1000	FAR CALL	NO	YES	YES	YES	FAR CALL
	0.1001	FAR JMP	NO	YES	YES	NO	-
	0.1010	SPECIAL; EMULATOR EXECUTION, SUPPLY EXTRA INSTRUCTION DATA ^a	NO	YES	NO	NO	-
	0.1011	ABORT PROFILE COLLECTION	NO	NO	NO	NO	-
	0.1100	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT W/PROBE (GRP 0)	NO	YES	YES	YES	EMULATOR PROBE
	0.1101	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT (GRP 0)	NO	YES	YES	NO	-
	0.1110	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT W/PROBE (GRP 1)	NO	YES	YES	YES	EMULATOR PROBE
	0.1111	x86 SYNCHRONOUS/ASYNCHRONOUS INTERRUPT (GRP 1)	NO	YES	YES	NO	-
404 CONVERTER (NEAR EDGE ENTRY)	1.0000	IP-RELATIVE JNZ FORWARD (OPCODE: 75, OF 85)	NO	YES	YES	NO	-
	1.0001	IP-RELATIVE JNZ BACKWARD (OPCODE: 75, OF 85)	NO	YES	YES	YES	JNZ
	1.0010	IP-RELATIVE CONDITIONAL JUMP FORWARD - (JCC, JCXZ, LOOP)	NO	YES	YES	NO	-
	1.0011	IP-RELATIVE CONDITIONAL JUMP BACKWARD - (JCC, JCXZ, LOOP)	NO	YES	YES	YES	COND JUMP
	1.0100	IP-RELATIVE, NEAR JMP FORWARD (OPCODE: E9, EB)	NO	YES	YES	NO	-
	1.0101	IP-RELATIVE, NEAR JMP BACKWARD (OPCODE: E9, EB)	NO	YES	YES	YES	NEAR JUMP
	1.0110	RET/RET IMM16 (OPCODE C3, C2 M)	NO	YES	YES	NO	-
	1.0111	IP-RELATIVE, NEAR CALL (OPCODE: E8)	NO	YES	YES	YES	NEAR CALL
	1.1000	REPE/REPNE CMPS/SCAS (OPCODE: A6, A7, AE, AF)	NO	YES	NO	NO	-
	1.1001	REP MOV/S/STOS/LDOS (OPCODE: A4, A5, AA, AB, AC, AD)	NO	YES	NO	NO	-
	1.1010	INDIRECT NEAR JMP (OPCODE: FF /4)	NO	YES	YES	NO	-
	1.1011	INDIRECT NEAR CALL (OPCODE: FF /2)	NO	YES	YES	YES	NEAR CALL
	1.1100	LOAD FROM I/O MEMORY (TLB.ASI!=0) (NOT USED IN T1)	NO	YES	NO	NO	-
	1.1101	AVAILABLE FOR EXPANSION	NO	NO	NO	NO	-
	1.1110	DEFAULT CONVERTER EVENT; SEQUENTIAL 406	NO	NO	NO	NO	-
	1.1111	NEW PAGE (INSTRUCTION ENDS ON LAST BYTE OF A PAGE FRAME OR STRADDLES ACROSS A PAGE FRAME BOUNDARY) 408	NO	YES	NO	NO	-

FIG. 4B









EVENTS
 pe_{init} = "initiate packet" profile event
 pe_{init} = non-"initiate packet" profile event
 pe = any profile event
 te = timer expiry
 ap = abort packet

STATE VARIABLES
 PR = Profile_Request flag
 PA = Profile_Active flag

RULES
 te EVENT:
 PR ← 1
 PR & PA & pe_{init}:
 PR ← 0
 PA ← 1
 Init Packet_Reg
 Save Timestamp
 Log Event (CAP)
 Full Packet? / Packet_Reg++

PA & pe:
 Log Event (CAP or NE)
 Full Packet? / Packet_Reg++

FULL PACKET:
 PA ← 0
 profile exception

ap EVENT:
 PA ← 0

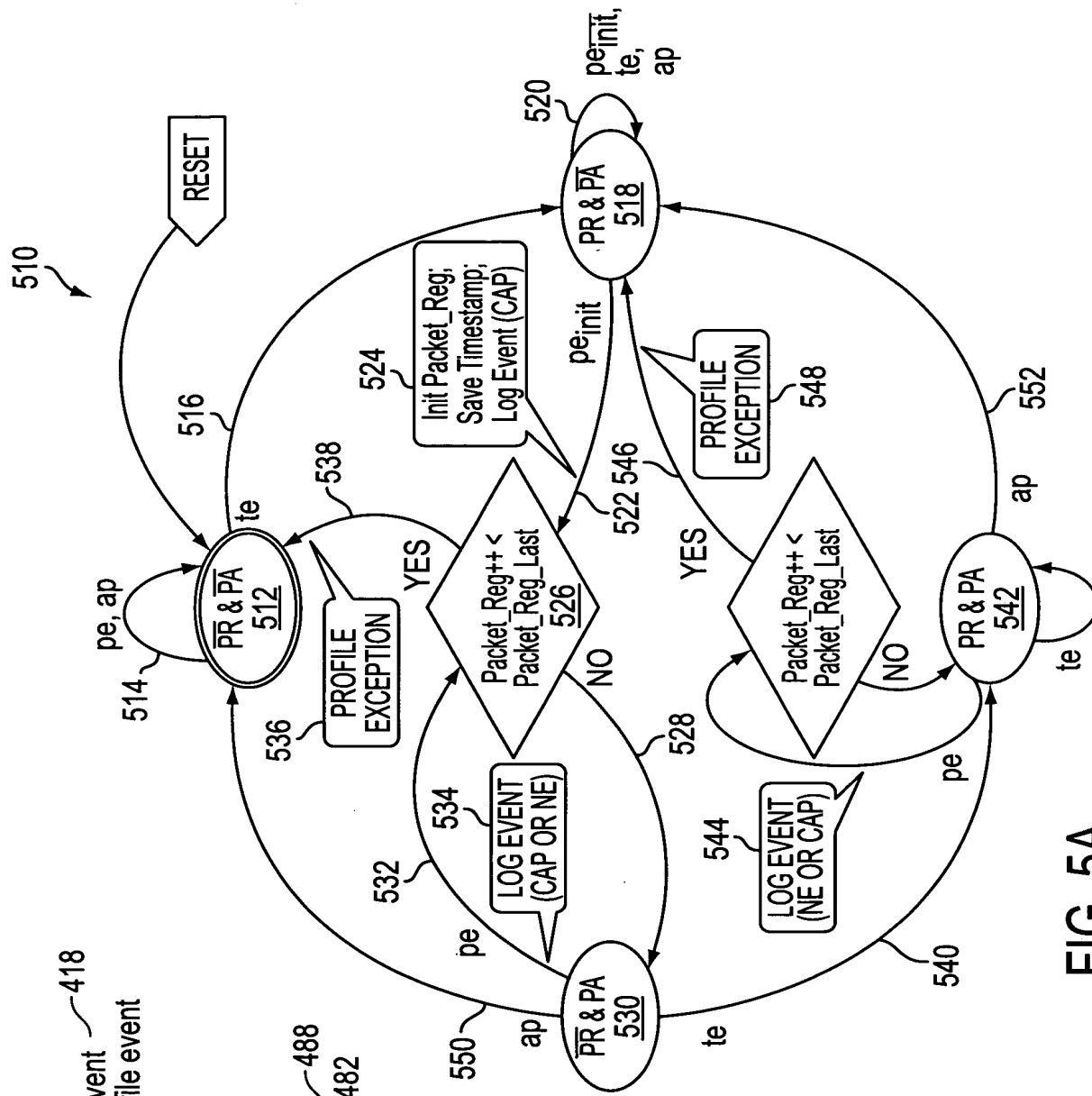
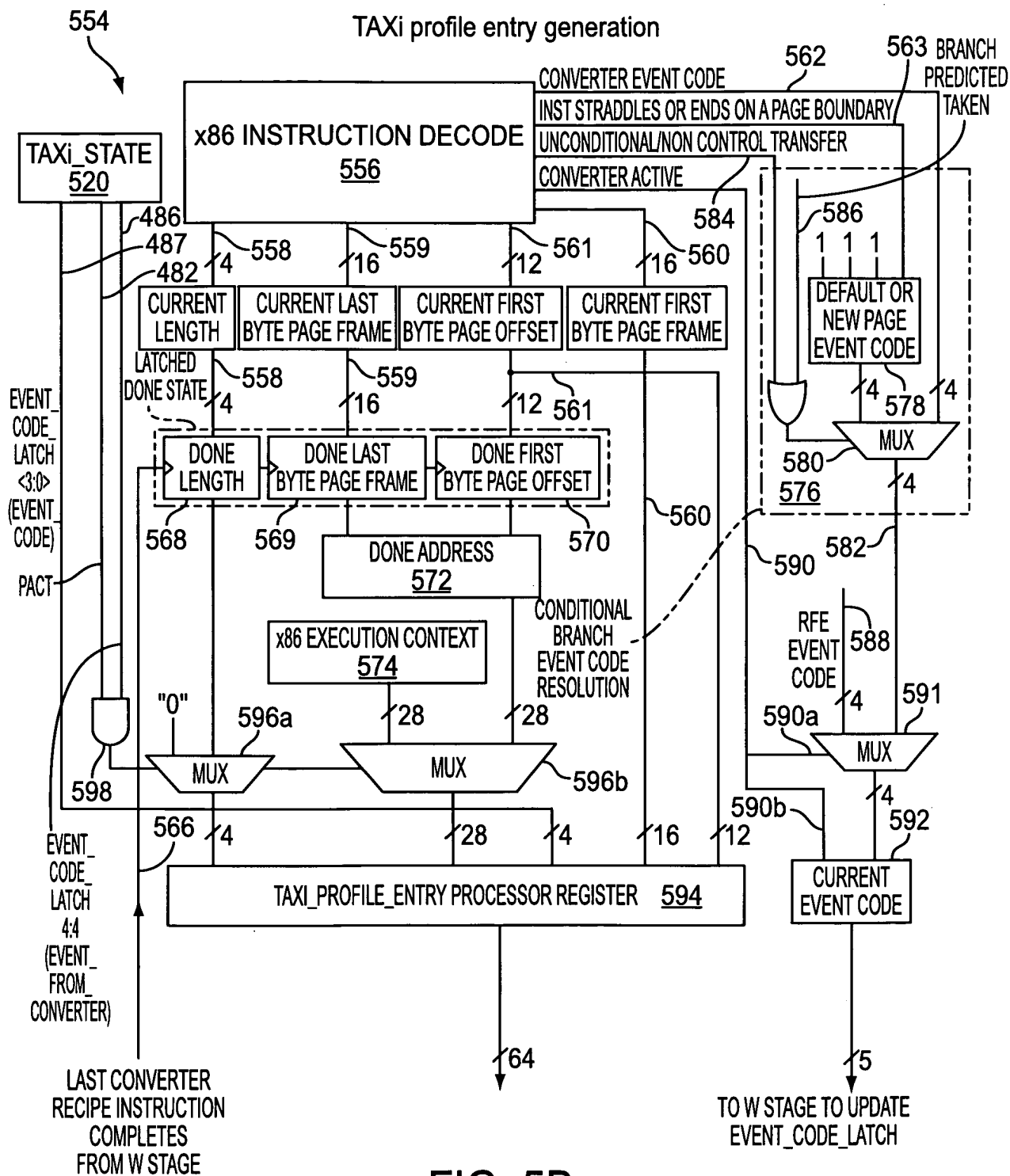


FIG. 5A



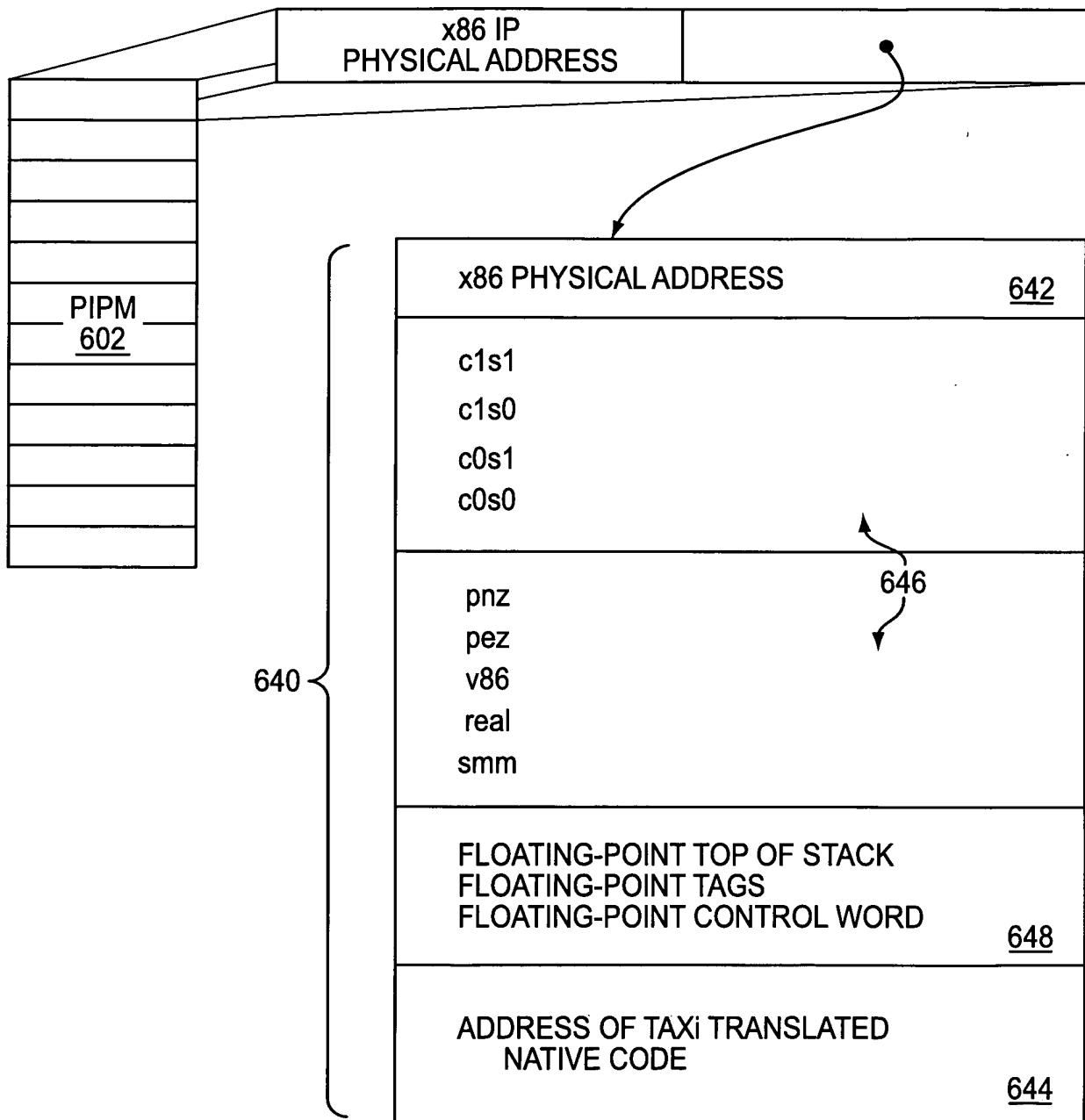


FIG. 6A

EVENT CODE FROM RFE RESTARTING CONVERTER
OR MAPPING OF CONVERTER'S x86 OPCODE

RFE OR PREVIOUS CONVERTER CYCLE

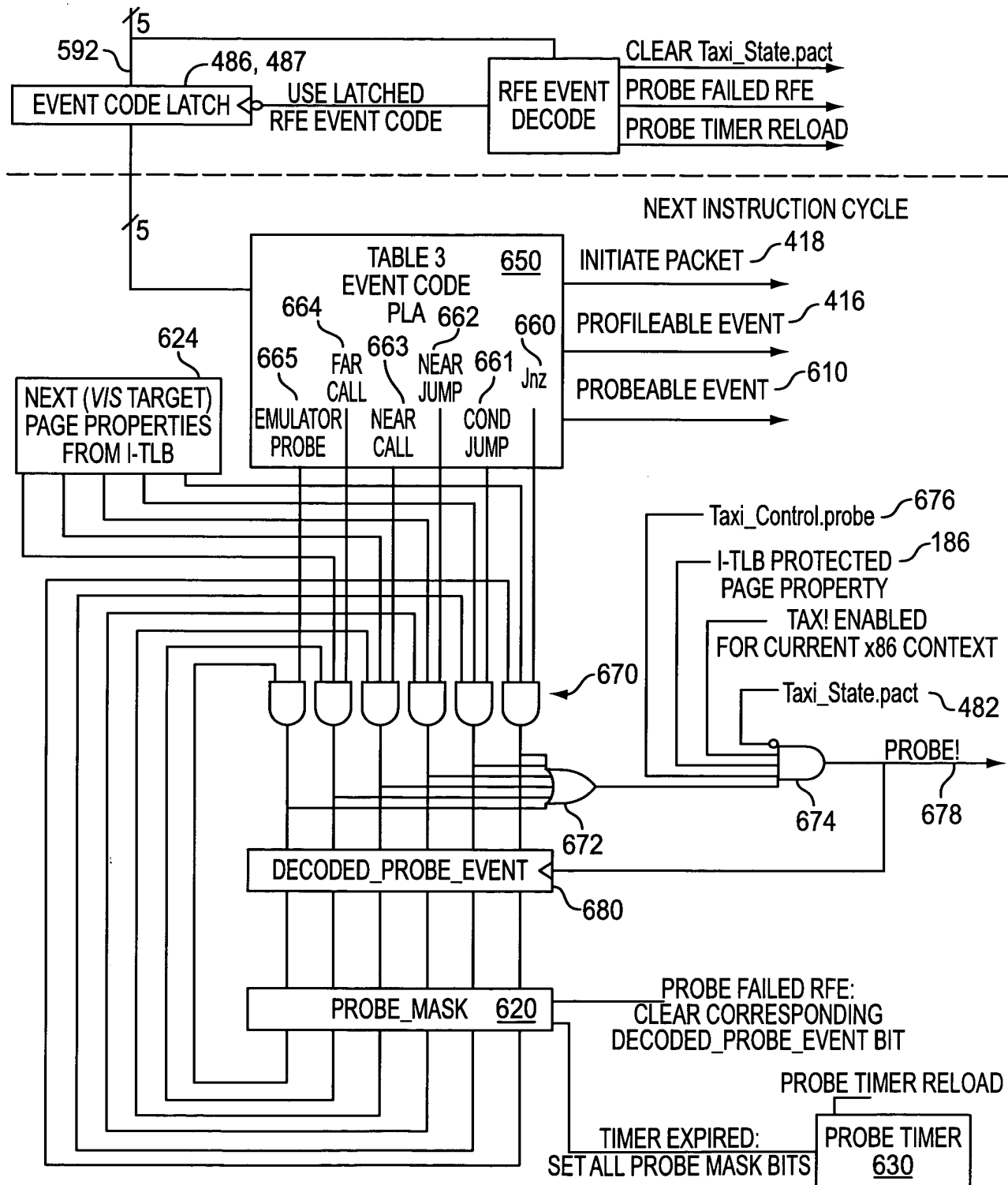


FIG. 6B

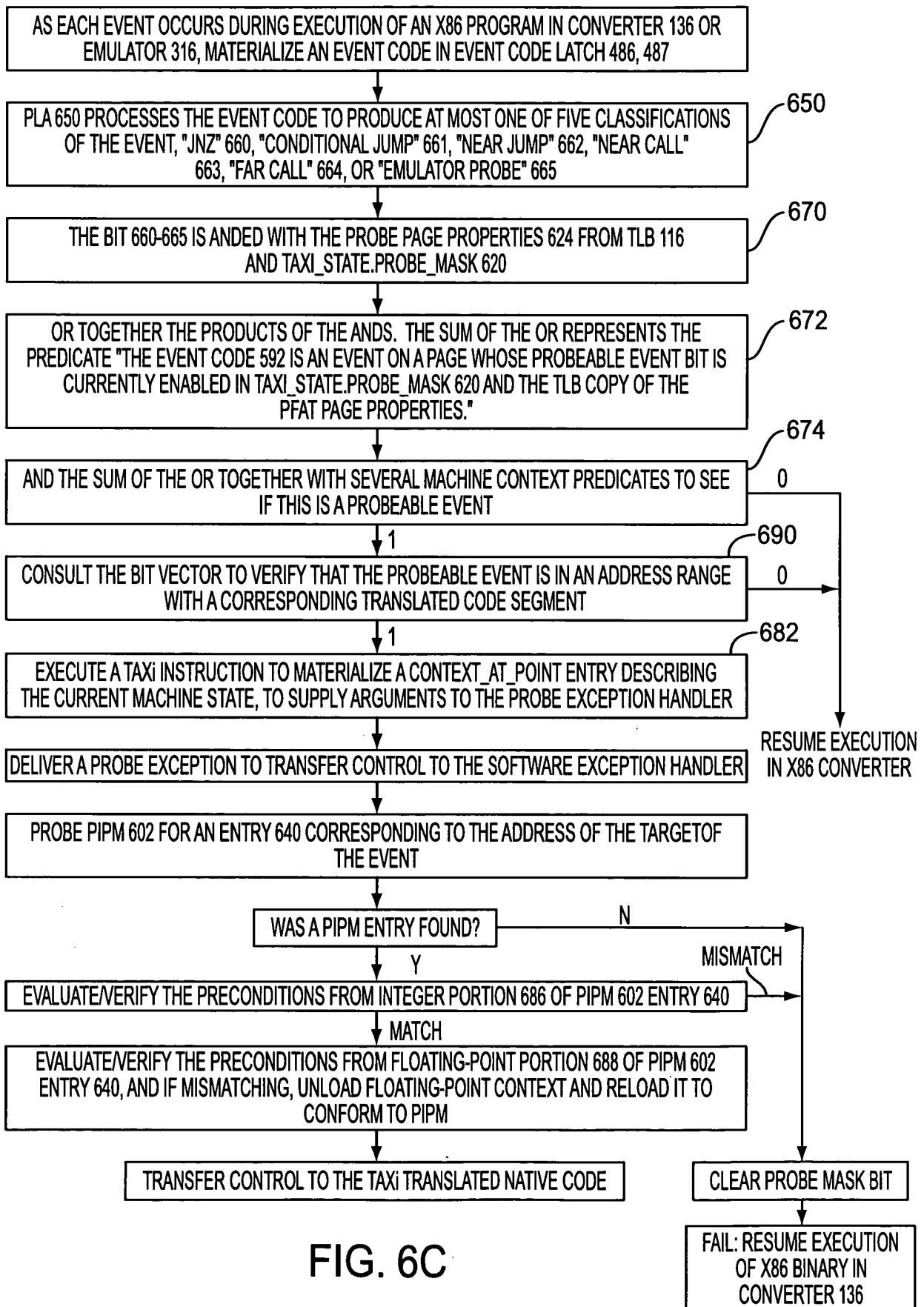


FIG. 6C

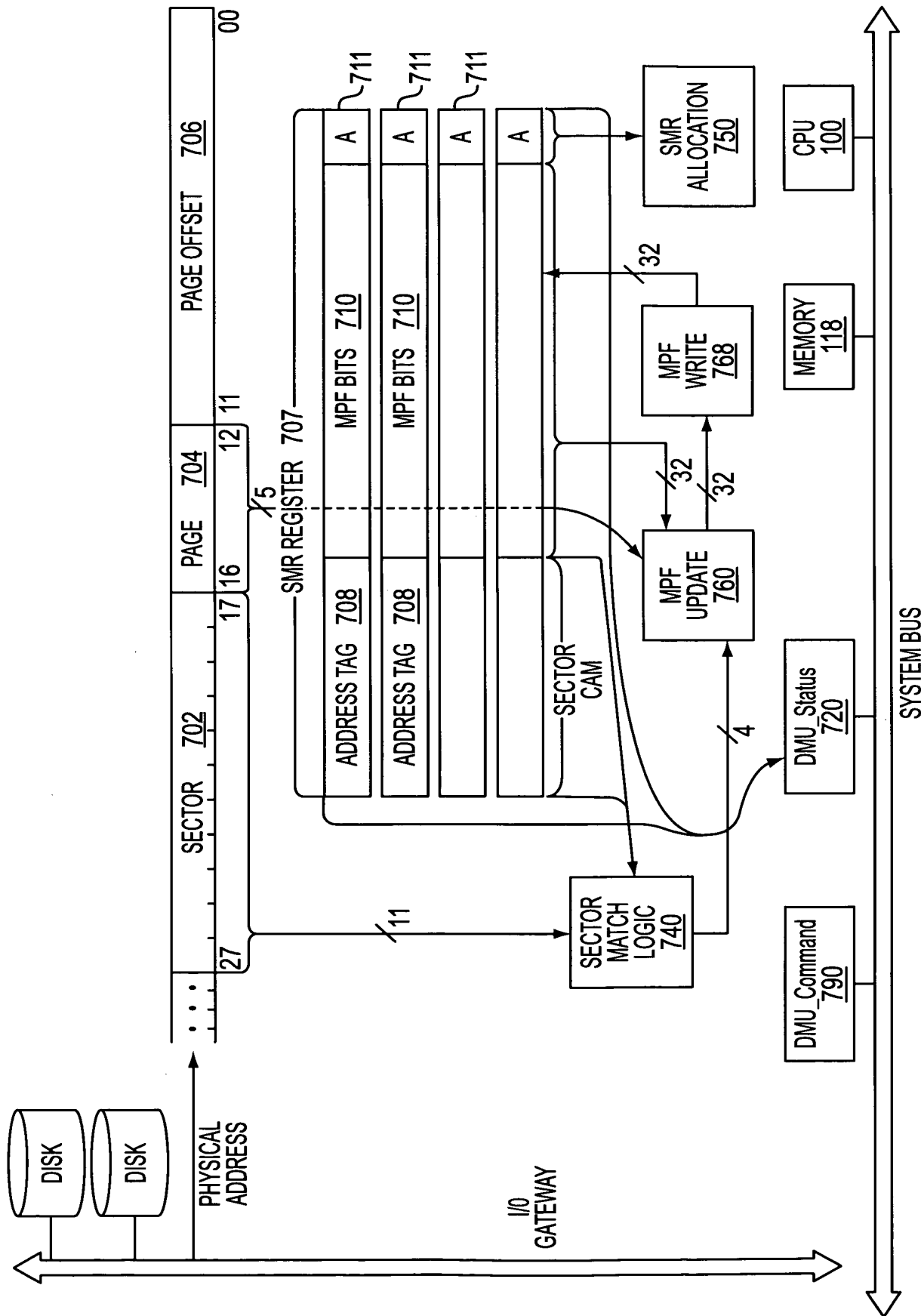


FIG. 7A

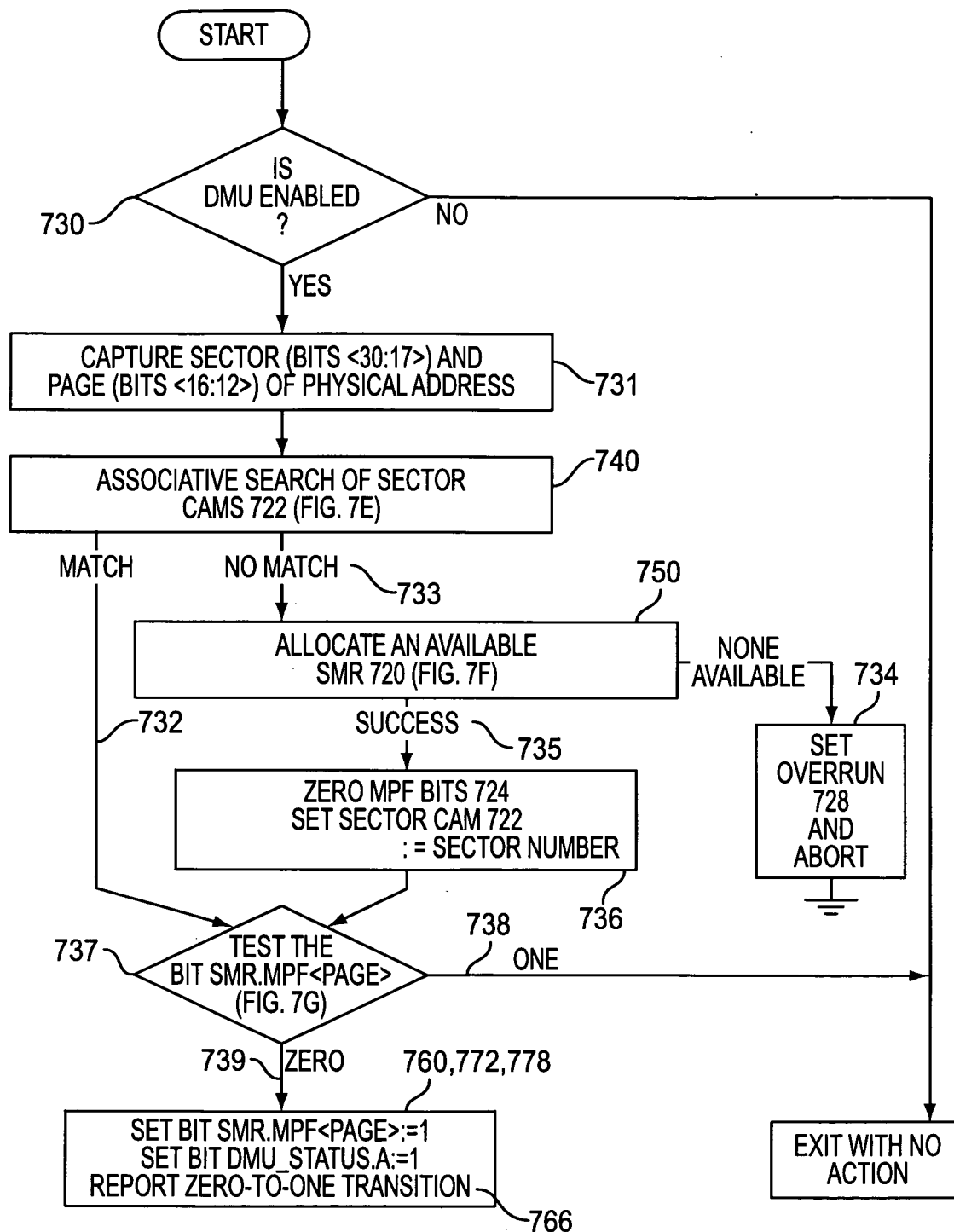
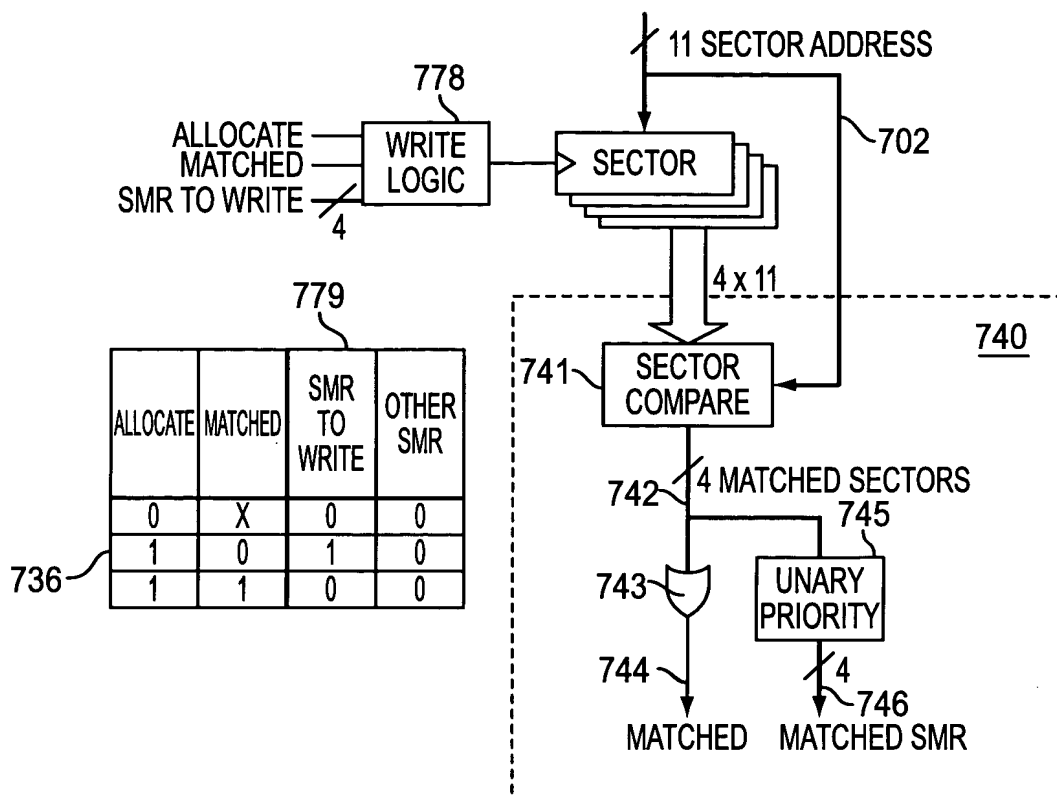
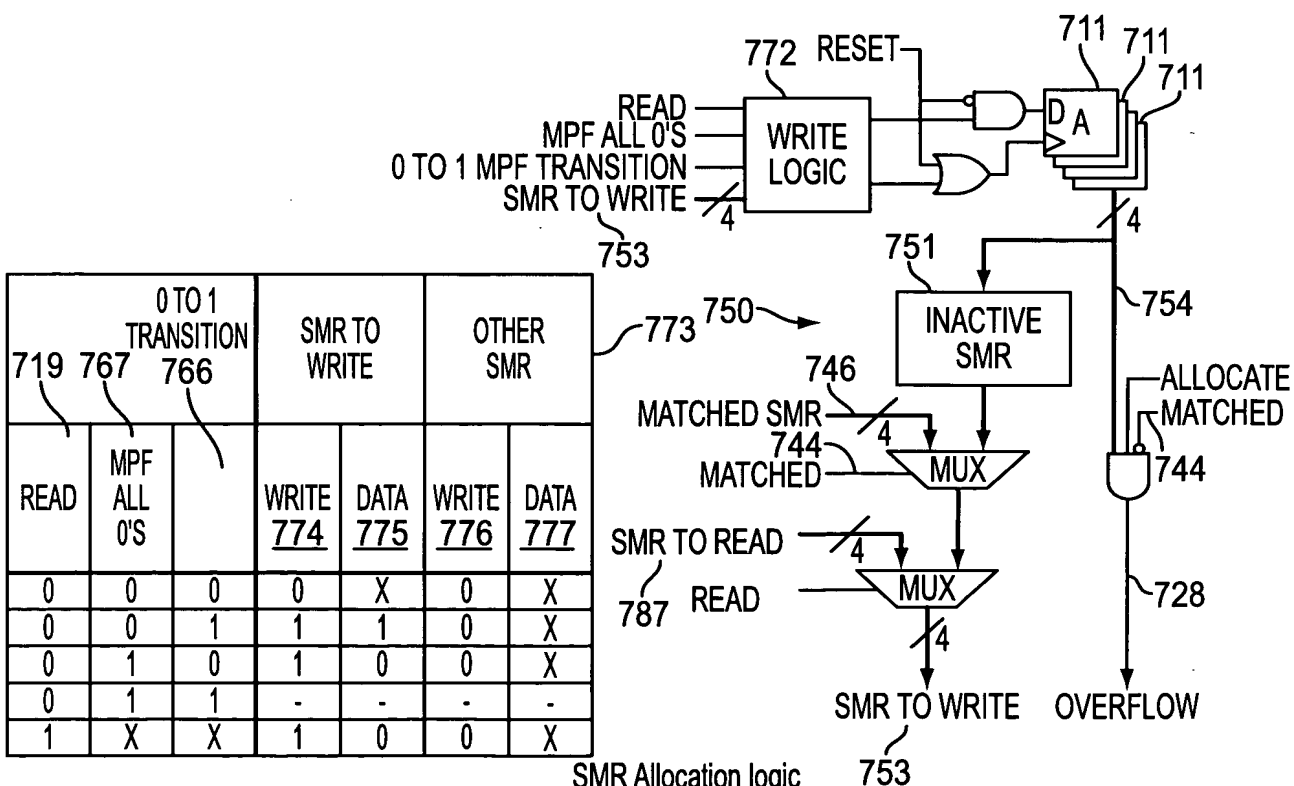


FIG. 7D



Sector match logic

FIG. 7E

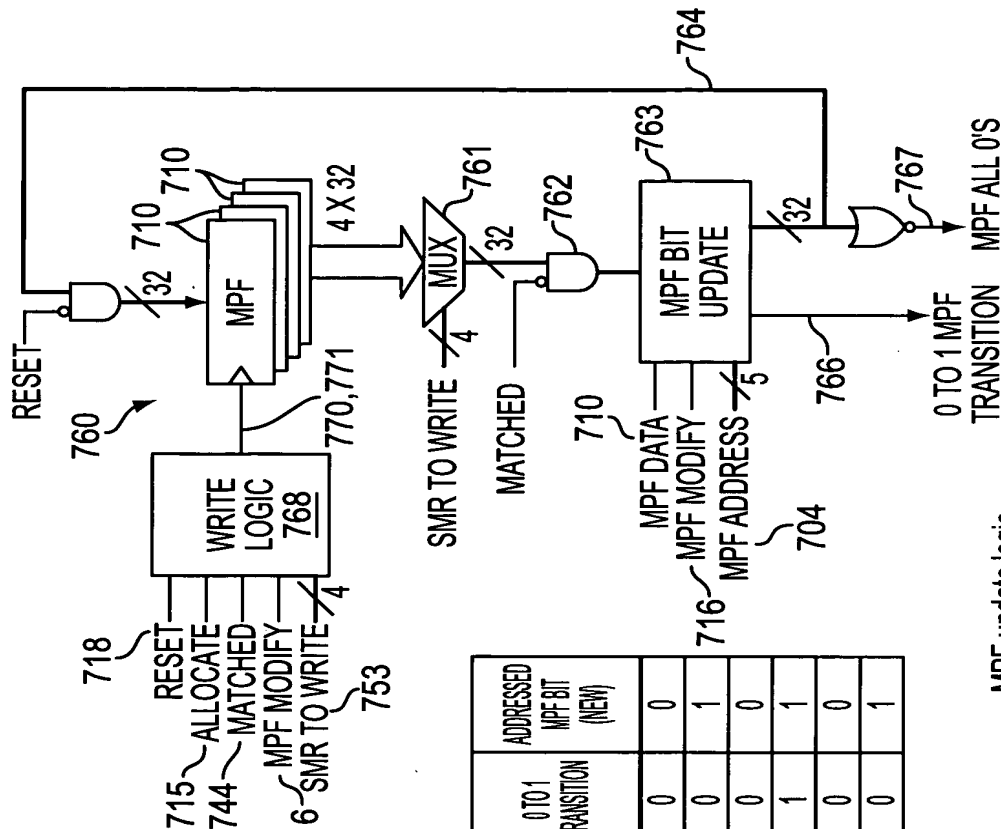


SMR Allocation logic

FIG. 7F

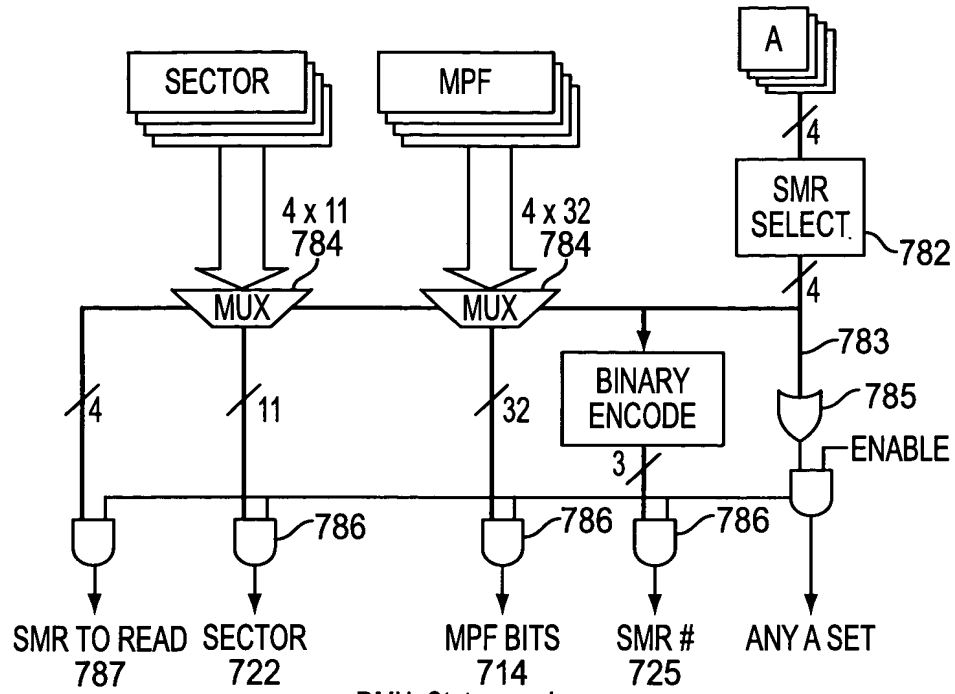
718	715	744	716	770	771
RESET	ALLOCATE	MATCHED	MPF MODIFY	SMR TO WRITE	OTHER TO SMR
0	X	X	0	0	0
0	0	0	1	0	0
0	X	1	1	1	0
0	1	X	1	1	0
1	X	X	X	1	1

MPF MODIFY	ADDRESSED MPF BIT (OLD)	MPF DATA (NEW)	0 TO 1 TRANSITION	ADDRESSED MPF BIT (NEW)
0	0	X	0	0
0	1	X	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	0	1



MPF update logic

FIG. 7G



DMU_Status read
FIG. 7H

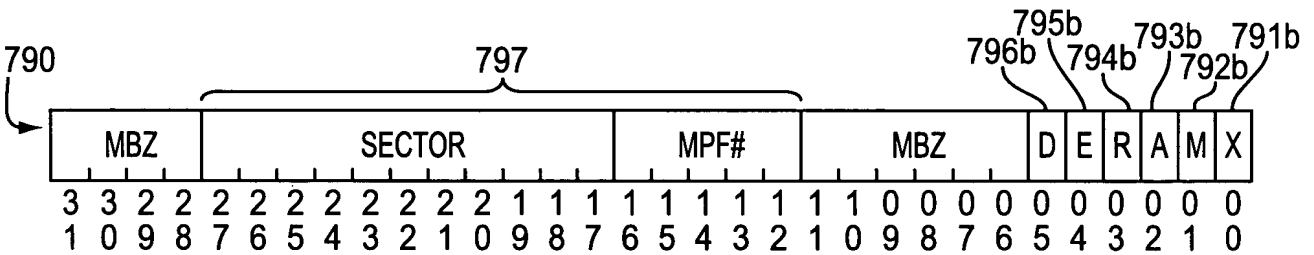
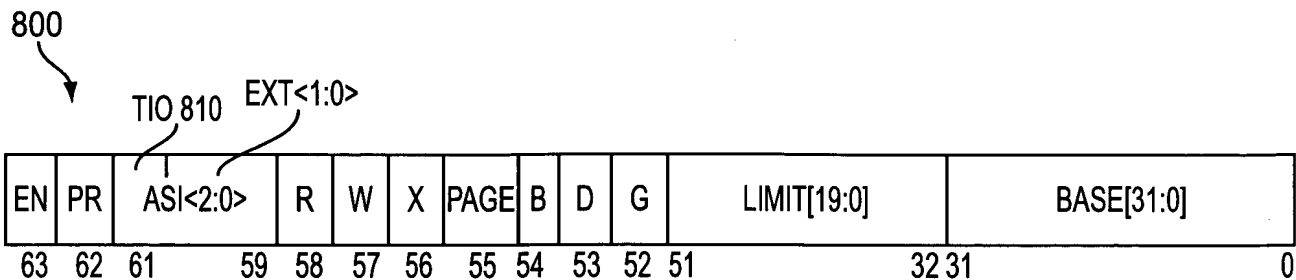


FIG. 7I

COMMAND BIT	BIT POSITION	MEANING
D	5	DISABLE MONITORING OF DMA WRITES BY CLEARING THE DMU ENABLE FLAG
E	4	ENABLE MONITORING OF DMA WRITES BY SETTING THE DMU ENABLE FLAG
R	3	RESET ALL SMRS: CLEAR ALL A AND MPF BITS AND CLEAR THE DMU OVERRUN FLAG
A	2	ALLOCATE AN INACTIVE SMR ON A FAILED SEARCH
M	1	ALLOW MPF MODIFICATIONS
X	0	NEW MPF BIT VALUE TO RECORD ON SUCCESSFUL SEARCH (OR ALLOCATION)

M	X	ACTION
0	-	INHIBIT MODIFICATION OF THE MPF BIT
1	0	CLEAR THE CORRESPONDING MPF BIT
1	1	SET THE CORRESPONDING MPF BIT

FIG. 7J



SIZE	BIT(S)	NAME	FUNCTION
1	63	SEG.EN	ENABLES SEGMENT LIMIT/PROTECTION CHECKING
1	62	SEG.PR	CHOOSES WHICH PROTECTION BITS TO USE FOR PAGE TABLE PROTECTION - (0 MEANS PSW.UK OR 1 MEANS MISC.UK)
3	61:59	SEG.AS	ADDRESS SPACE (ONLY USED WHEN SEG.PAGE IS 0)
		SEG.TIO, SEG.EXT	ADDRESS SPACE EXTENSION (ONLY USED WHEN SEG.PAGE IS 1)
3	58:56	SEG.RWX	READ/WRITE/EXECUTE '1' MEANS ENABLED - ALL 000 MEANS IT'S AN INVALID SEGMENT
1	55	SEG.PAGE	ENABLES THE PAGING SYSTEM -- (TRANSLATION AND CHECKING)
1	54	SEG.B	SEGMENT SIZE (1 MEANS 32-BIT, 0 MEANS 16-BIT)
1	53	SEG.D	SEGMENT DIRECTION (0 MEANS EXPAND UP)
1	52	SEG.G	SIZE OF LIMIT (1 MEANS IT'S IN 4k PAGES)
20	51:32	SEG.LIMIT	SEGMENT LIMIT
32	31:0	SEG.BASE	SEGMENT BASE

FIG. 8A

AT CODE GENERATION TIME:

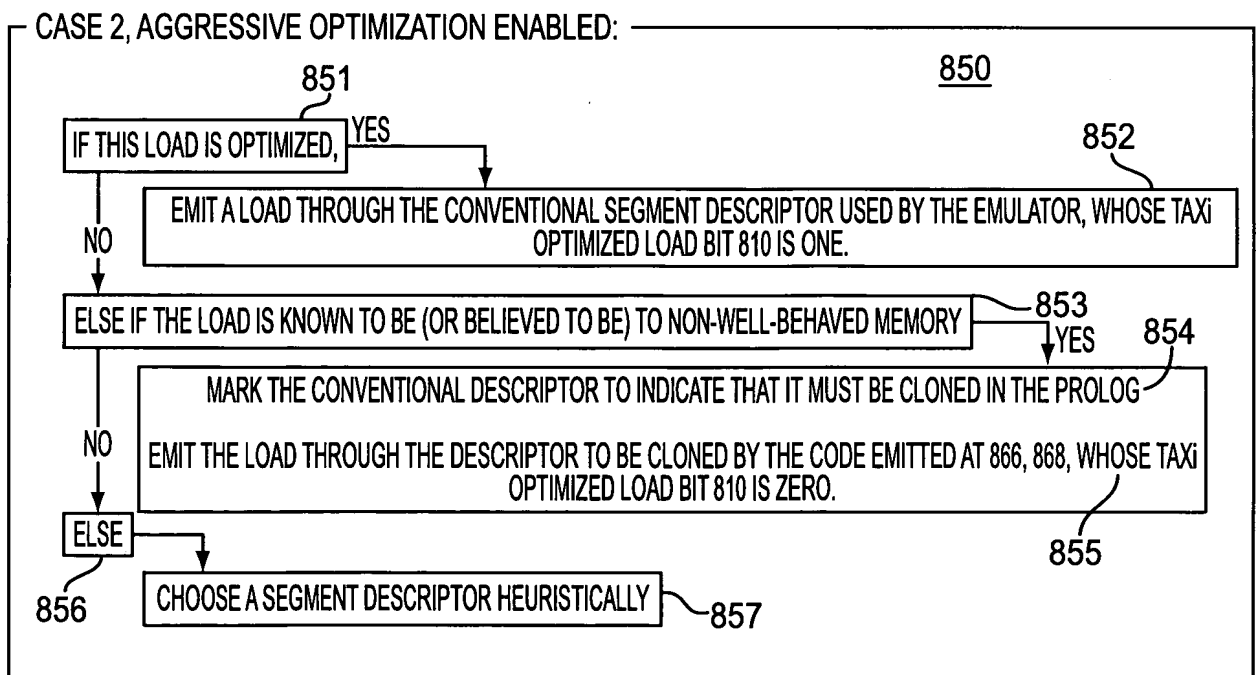
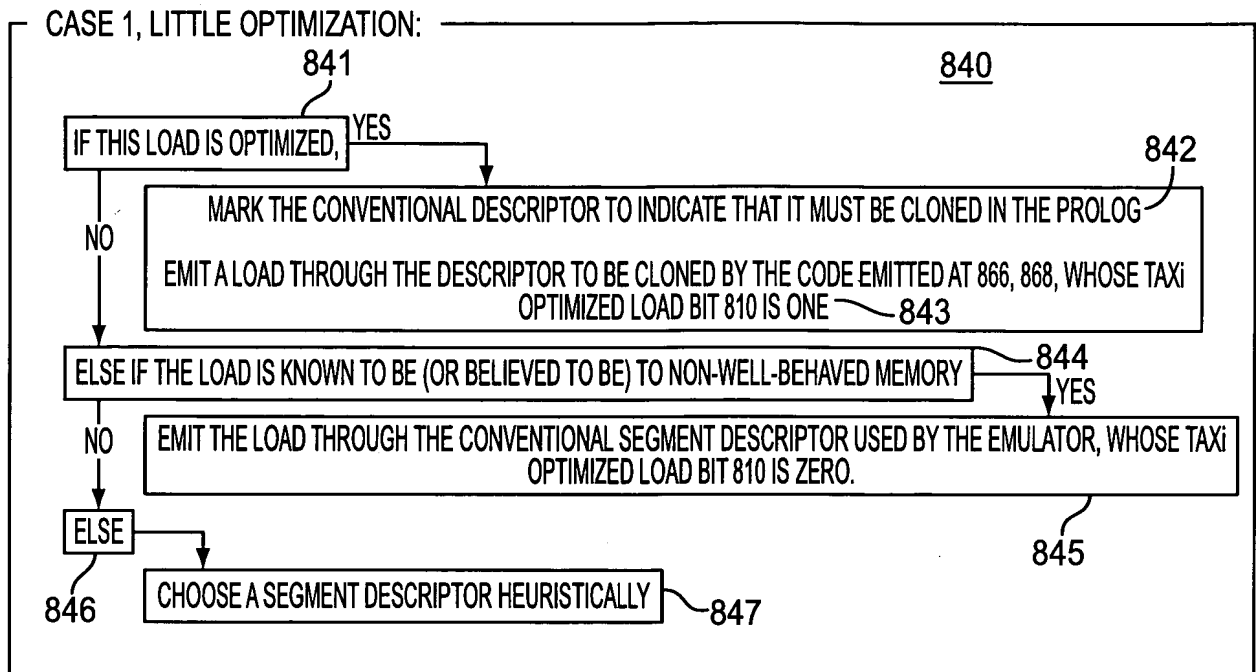


FIG. 8B

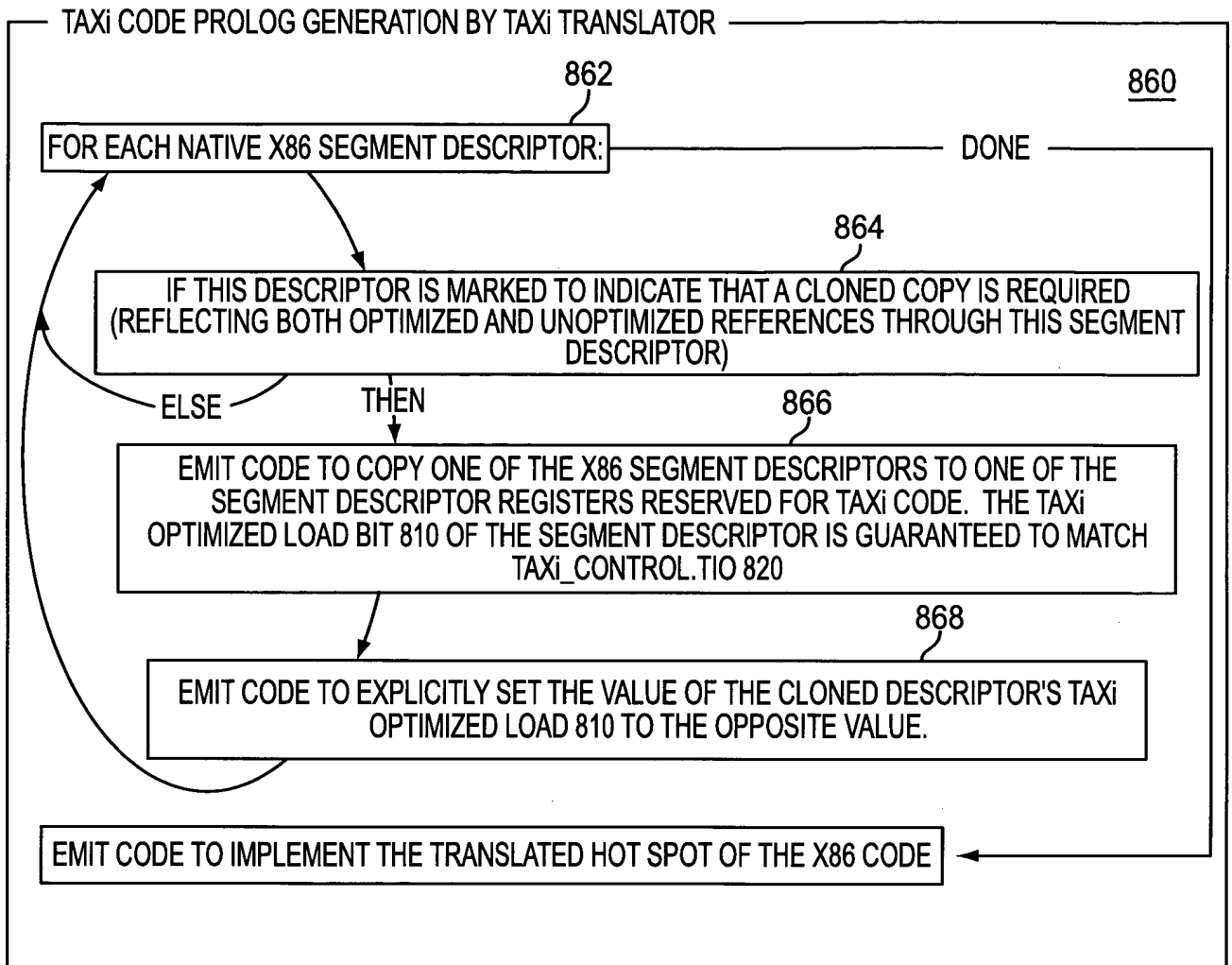


FIG. 8C

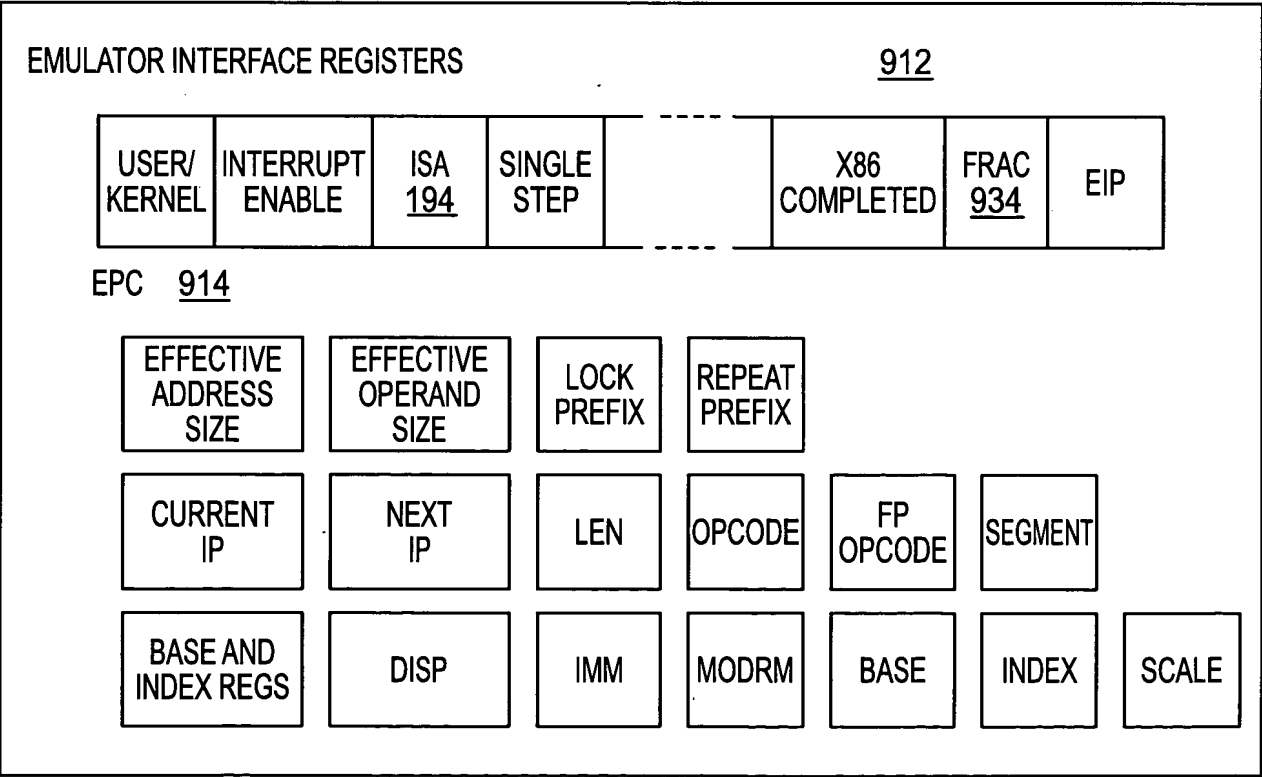
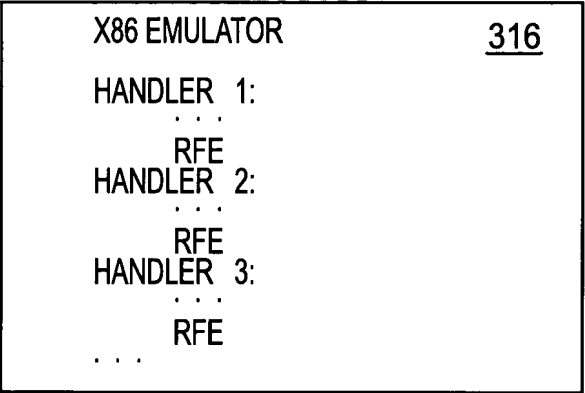
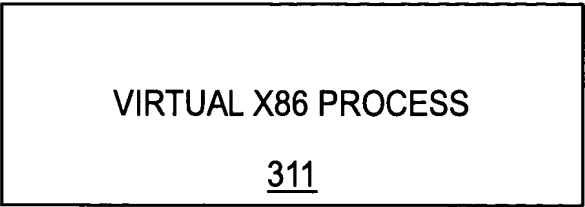
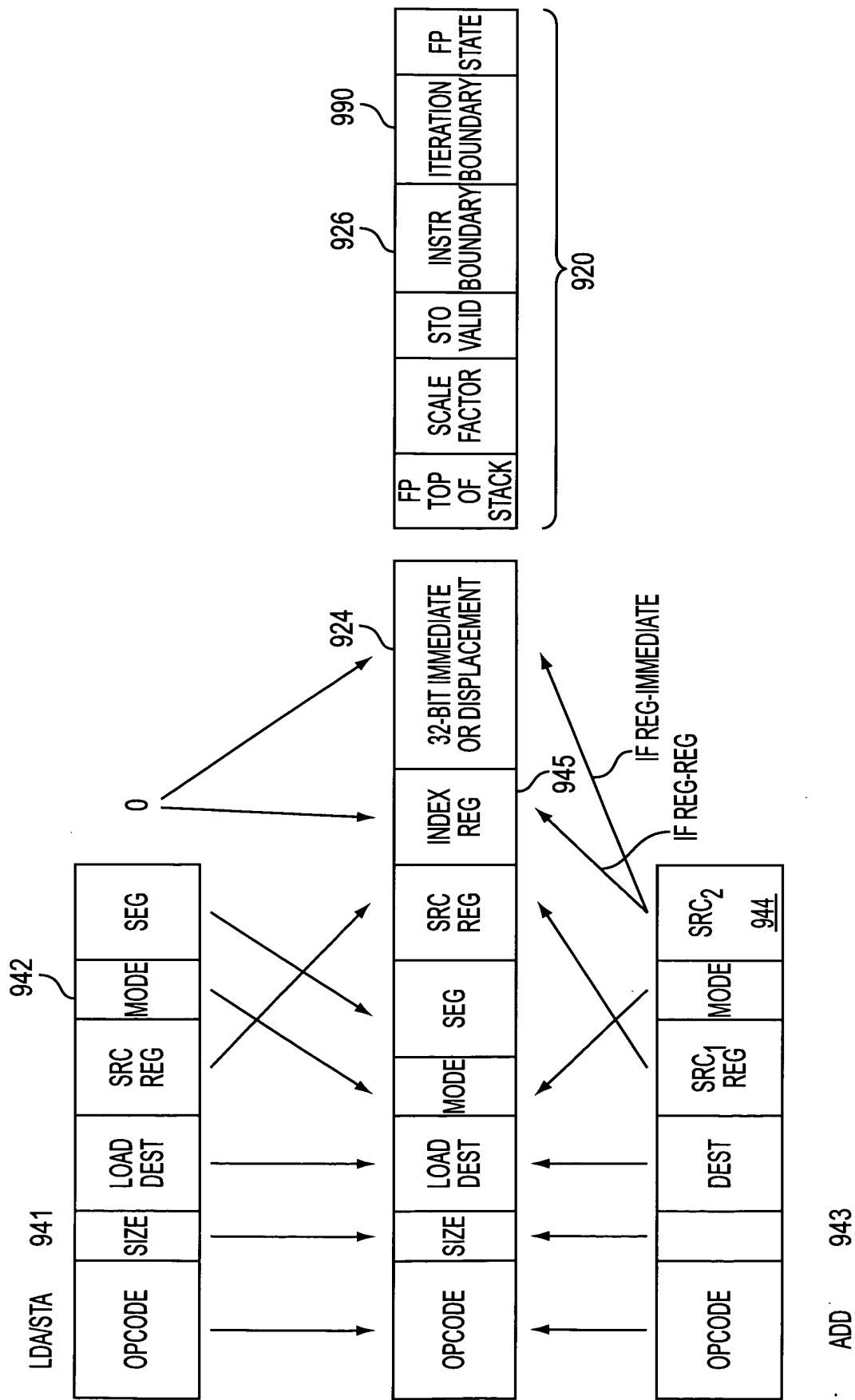


FIG. 9B



MNEMONIC	TYPE	DESCRIPTION OF SIDE-BAND INFORMATION
INSTRUCTIONS WITH Imm6 FIELD		THE CONVERTER MAY SUPPLY A FULL 32-BIT IMMEDIATE.
BRANCHES WITH DISPLACEMENT		THE CONVERTER MAY SUPPLY A FULL 32-BIT DISPLACEMENT.
LDA/STA	INTEGER	A FULL 32-BIT DISPLACEMENT IS SENT ON THE IMMEDIATE BUS; THIS IS ADDED TO SRC1 TO COMPUTE THE OFFSET FOR SOME ADDRESSING MODES.
CJcond	INTEGER	THE CONVERTER MAY SPECIFY A 16 OR 32-BIT ADDRESS SIZE IN PARALLEL WITH THIS INSTRUCTION (A 32-BIT DISPLACEMENT MAY ALSO BE PROVIDED).
CJcond	INTEGER	THE CONVERTER MAY SPECIFY A 16 OR 32-BIT ADDRESS SIZE IN PARALLEL WITH THIS INSTRUCTION. A 32-BIT DISPLACEMENT MAY ALSO BE PROVIDED.
FROMPR	INTEGER	3-BITS OF TOS (TOP-OF-STACK) ARE SENT ON THE IMMEDIATE BUS IN PARALLEL WITH THIS INSTRUCTION FOR USE BY THE FNSTSW INSTRUCTION CONVERTER SEQUENCE.
LEA	INTEGER	A 6-BIT INDEX REGISTER SPECIFIER, A 32- BIT DISPLACEMENT, AND A 2-BIT SCALE FACTOR ARE PASSED FROM THE CONVERTER AS ADDITIONAL INPUT TO THE HARDWARE IN ORDER TO FORM A COMPLETE x86 ADDRESSING MODE.
LDAI	INTEGER	A 6-BIT INDEX REGISTER SPECIFIER, A 32- BIT DISPLACEMENT, AND A 2-BIT SCALE FACTOR ARE PASSED FROM THE CONVERTER AS ADDITIONAL INPUT TO THE HARDWARE IN ORDER TO FORM A COMPLETE x86 ADDRESSING MODE. ADDITIONALLY, A SECOND DESTINATION REGISTER IS PASSED AS THE DESTINATION OF THE ADDRESS AUTOINCREMENT MODE.
LOOP, LOOPZ, LOOPNZ	INTEGER	THE CONVERTER MAY SPECIFY A 16 OR 32-BIT ADDRESS SIZE IN PARALLEL WITH THIS INSTRUCTION. A 32-BIT DISPLACEMENT MAY ALSO BE PROVIDED.
STAI	INTEGER	A 6-BIT INDEX REGISTER SPECIFIER, A 32- BIT DISPLACEMENT, AND A 2-BIT SCALE FACTOR ARE PASSED FROM THE CONVERTER AS ADDITIONAL INPUT TO THE HARDWARE IN ORDER TO FORM A COMPLETE x86 ADDRESSING MODE. ADDITIONALLY, A SECOND DESTINATION REGISTER IS PASSED AS THE DESTINATION OF THE ADDRESS AUTOINCREMENT MODE.
PSHUFW	MMX	ONLY 6 BITS OF THE Imm8 ARE STORED IN THE INSTRUCTION. THE REMAINING TWO BITS ARE CREATED BY THE HW CONVERTER.
FLDA	FP EP	A 6-BIT INDEX REGISTER SPECIFIER AND A 32- BIT DISPLACEMENT, AND A 2-BIT SCALE FACTOR ARE PASSED FROM THE CONVERTER AS ADDITIONAL INPUT TO THE HARDWARE IN ORDER TO FORM A COMPLETE x86 ADDRESSING MODE.
FTST	FP EP	1-BIT OF STO_VALID IS SENT ON THE IMMEDIATE BUS IN PARALLEL WITH THIS INSTRUCTION.
FSTA	FP EP	A 6-BIT INDEX REGISTER SPECIFIER AND A 2- BIT SCALE FACTOR ARE PASSED FROM THE CONVERTER AS ADDITIONAL INPUT TO THE HARDWARE IN ORDER TO FORM A COMPLETE x86 ADDRESSING MODE.
FXAM	FP EP	1 BIT STO_VALID IS PASSED ON THE IMMEDIATE BUS.
INSTRUCTION CONTROL		INSTRUCTION BOUNDARY INFORMATION: - START OF INSTRUCTION OR STRING ITERATION - LAST OF SEQUENCE - FP_DP/,,, INTERNMENT CONTROL - FP_TAG_MAP INTERNMENT CONTROL

FIG. 9D

X86 instruction PUSHAD

Temp := (ESP)
 Push(EAX)
 Push(ECX)
 Push(EDX)
 Push(EBX)
 Push(Temp)
 Push(EBP)
 Push(ESI)
 Push(EDI)

Native Instruction Recipe

954 MOV.64 tmp_d, ESP /* copy working SP to temp */
 951 STOREDEC.X EAX,SS,tmp_d
 STOREDEC.X ECX,SS,tmp_d
 STOREDEC.X EDX,SS,tmp_d
 STOREDEC.X EBX,SS,tmp_d
 STOREDEC.X ESP,SS,tmp_d
 STOREDEC.X EBP,SS,tmp_d
 STOREDEC.X ESI,SS,tmp_d
 955 MOV.64 EDI,SS,tmp_d
 952 ESP,tmp_d /* commit new SP */

950

FIG. 9E

IDIOM	USAGE
LOAD / OP [/STORE]	LOAD DATA
COMPLEX ADDRESS CALCULATION	COMPUTED OFFSET
MOV mem, [DEFGS]S / PUSH [DEFGS]S (SELECTOR PUSH/STORE)	SELECTOR (PROCESSOR REGISTER NOT DIRECTLY ACCESSIBLE BY STORE INSTRUCTIONS)
PUSHA (PUSH ALL)	INTERMEDIATE STACK POINTER; COMMIT AT END
POPA (POP ALL)	INTERMEDIATE STACK POINTER; COMMIT AT END
MOV mem, lmm / PUSH lmm	INTERMEDIATE (NOT AVAILABLE AS AN OPERAND TO STORE INSTRUCTION)
MULTIPLY	INTERMEDIARY TO CONNECT CONTIGUOUS NATIVE REGISTER PAIR TO X86 REGISTER PAIR
DIVIDE	
XCHG	THE CLASSIC USE OF A TEMPORARY!
POP mem	STACK POINTER UNTIL MEMORY OPERATIONS ARE FINISHED

FIG. 9F

X86 instruction ADD r/m8,r8
 DEST := DEST + SRC;

Native Instruction Recipe

962 LDA.b.write_intent tmp_d,Seg.Base,Base
 963 ADD.b tmp_d,tmp_d,reg
 964 STA.b tmp_d,Seg.Base,Base

960

961

FIG. 9G

967
 X86 instruction CALL r/mX /* near absolute call */
 IF target instruction pointer is not within code segment limit
 THEN #GP(0); FI; 968
 IF stack not large enough for a 4-byte return address
 THEN #SS(0); FI; 969
 Push(EIP);
 EIP := EIP + DEST;

FIG. 9H

976
 X86 instruction CALL re1X /* near IP-relative call */
 IF target instruction pointer is not within code segment limit
 THEN #GP(0); FI;
 IF stack not large enough for a 4-byte return address
 THEN #SS(0); FI;
 Push(EIP);
 EIP := EIP + DEST;

FIG. 9I

980
 X86 instruction LOOP imm8
 Count := ECX;
 Count := Count - 1;
 IF (Count == 0)
 THEN BranchCond := 1;
 ELSE BranchCond := 0;
 FI;

IF (BranchCond == 1)
 THEN
 NextEIP := NextEIP + SignExtend(DEST);
 IF target instruction pointer is not with code segment limit
 THEN
 #GP(0); /* ECX not modified */
 ELSE
 ECX := COUNT;
 EIP := NextEIP;
 FI;
 ELSE
 ECX := Count;
 Terminate loop and continue program execution at EIP;
 FI;

FIG. 9J

970
 Native Instruction Recipe
 LOAD.limit_check r0,CS:reg_d
 971
 972
 STOREDEC.X IP,SS,ESP
 JR reg_d
 973

Native Instruction Recipe
 977
 STOREDEC.X IP,SS,ESP
 JR reg_d
 978

981
 Native Instruction Recipe
 DEC.X ECX,ECX
 982
 CJNE ECX,r0,imm8
 983

986
 X86 REPNZ MOVs
 WHILE ECX ≠ 0
 DO
 service pending interrupts (if any);
 execute associated MOV instruction;
 ECX := ECX - 1;
 IF ECX = 0
 THEN exit WHILE loop;
 IF ZF = 1
 THEN exit WHILE loop;
 FI;
 OD;

988 { LDA.b
 STOREINC
 989 { JNZ
 991 { LDA.b
 STOREINC
 989 { JNZ
 LDA.b
 STOREINC
 989 { JNZ
 ...
 ...

987
 tmp_d, src++
 dest++, tmp_d
 predicted not taken
 tmp_d, src++
 dest++, tmp_d
 predicted not taken
 tmp_d, src++
 dest++, tmp_d
 predicted not taken

FIG. 9K